MASTER EN SCIENCES INFORMATIQUES

Détection de fichiers pertinents
Experimentation sur l'analyse de traces

Drioul, Thomas; RAPPE, PIERRE-ANTOINE

Award date:
2014

Awarding institution:
Universite de Namur

Université de Namur
Faculté d'informatique
Année académique 2013–2014

Détection de fichiers pertinents : Expérimentation sur l'analyse de traces

Thomas Drioul
Pierre-Antoine Rappe

Maîtres de stage : Yann-Gaël Guéhéneuc

Promoteur : _____ (Signature pour approbation du dépôt - REE art. 40)
Naji Habra

Mémoire présenté en vue de l'obtention du grade de
Master en Sciences Informatiques.

Abstract

Résumé Depuis de nombreuses années, des recherches ont été effectuées pour mieux appréhender les concepts de maintenance logicielle et de compréhension logicielle. Ce travail s'inscrit dans cette ligne ayant pour finalité, la réduction du temps de maintenance. Cette réduction doit s'effectuer sur le temps de recherche car des études ont prouvé que le temps de recherche est bien plus important que le temps de modification lors d'une maintenance. Pour essayer de réduire ce temps, une expérimentation a été conduite pour déceler les fichiers pertinents. Une distinction a été faite entre les fichiers hautement et significativement pertinents. Cette expérimentation a donné des traces vidéos de résolutions de différentes tâches qui ont été analysées. Des critères de pertinence ont pu être en extraits : le temps total passé dans un fichier, le temps total de recherche passé dans un fichier et le nombre de revisites d'un fichier.

Mots clés Maintenance logicielle, compréhension logicielle, significativement pertinent et hautement pertinent.

Abstract For many years, researchs have been conducted to understand in a better way the concepts of software maintenance and software understanding. This work is axed on these fields in which a reduction of the maintenance time is the ultimate goal. Moreover, the reduction must be made on the search time because a statement is that the search time is more important than the edit time for a task. In order to reduce this time, an expriment has been conducted to find the pertinent files. A distinction has been made between a significantly and a highly relevant file. This experiment gave from video traces of resolved maintenance tasks. Some criteria have been extracted of these traces. These criteria are the total time spent in a file, the total searching time spent in a file and the revisit number of a file.

Keywords Software maintenance, software comprehension, significantly relevant and highly relevant.

Remerciements

Ce mémoire est le résultat de notre stage de 4 mois au sein du laboratoire Ptidej de l'Ecole Polytechnique de Montréal, Canada. Ce laboratoire a pour but de développer des théories, méthodes et outils pour évaluer et augmenter la qualité de programmes orientés objets. Ce stage fut supervisé par Yann-Gaël Guéhéneuc, Foutse Khomh et Zéphyrin Soh. Nous tenons particulièrement à les remercier pour nous avoir permis d'effectuer ce stage. Nous tenons aussi à les remercier pour la découverte qu'ils nous ont offerte du monde scientifique. Nous remercions aussi l'ensemble des chercheurs de l'équipe du laboratoire Ptidej et du laboratoire SoccerLab de l'école polytechnique de Montréal pour leurs conseils et leur sympathie lors de notre stage.

Nous tenions aussi à remercier notre promoteur de notre mémoire, Naji Habra, pour le suivi et les conseils donnés lors de notre stage et pour l'aide apportée lors de la rédaction de ce mémoire.

Notre mémoire se basant sur une expérimentation, nous remercions l'ensemble des sujets y ayant participé. Sans eux, nous ne pourrions avoir écrit ce travail.

Finalement, nous remercions nos parents, familles et amis de nous avoir permis d'accomplir ce stage à l'étranger mais aussi pour leur soutien lors de nos études et de la rédaction de ce mémoire.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

Depuis de nombreuses années, des recherches ont été effectuées pour mieux comprendre les concepts de maintenance logicielle et de compréhension logicielle.

1.1 Mise en contexte du travail

Nous allons définir brièvement ces 2 concepts qui sont à la base de ce mémoire. La maintenance logicielle est présentée à la section 1.1.1 et la compréhension logicielle à la section 1.1.2.

1.1.1 Maintenance logicielle

De nos jours, la plupart des modèles de cycle vie d'un logiciel montrent que la phase de maintenance devient de plus en plus importante et qu'elle consomme plus de la moitié du temps de vie d'un logiciel. Nous en distinguons 3 types principaux et 2 types secondaires [1][4, 27, 32, 23, 1, 16].

Maintenance adaptative Maintenance effectuée après un changement dans les exigences.

Maintenance corrective Maintenance effectuée après la découverte d'un bug.

Maintenance perfective Maintenance effectuée après un changement dans les besoins des consommateurs.

Maintenance préventive Maintenance effectuée pour pallier la menace potentielle d'une erreur ainsi que pour faciliter les maintenances future et la compréhension du système.

Maintenance d'urgence Maintenance effectuée quand un aléa apparait alors qu'une maintenance corrective est déjà en-cours.

1. Ces types de maintenance sont expliqués plus en détail à la section 3.3

La maintenance a un coût tant financier que temporel. Ce coût de maintenance surpasse le coût du développement sur ces 2 points. Les entreprises veulent donc diminuer le coût de maintenance. De nombreuses recherches sont menées dans ce but. Bien que des recherches aient été conduites, les coûts financiers n'ont pu être réduits de manière significative[1] et restent compris, la plupart du temps, entre 60 et 80% de l'argent dépensé pendant la vie totale d'un système d'informations.

Dans une maintenance logicielle, différents modèles sont utilisés pour donner un caractère automatique à la réalisation d'une tâche de maintenance.

1.1.2 Compréhension du logiciel

Le logiciel existant doit être compris avant de pouvoir effectuer une modification en accord avec un plan de maintenance. Réduire cette phase de compréhension permettrait de réduire le temps global de maintenance.

De nombreux outils tentent de fournir des informations de compréhension logicielle en permettant la visualisation de données statiques ou dynamiques du programme étudié.

Les outils fournissent aux développeurs des informations, des modules, des fichiers, des classes qui leur permettraient de comprendre plus vite le logiciel pour entamer au plus tôt la phase d'implémentation.

Dans ce mémoire, nous utiliserons la notion de pertinence de fichier donnée par Lee et Kang[37]. Les auteurs distinguent deux types de pertinence d'entité. Une entité peut être un élément Java ou une ressource. Un élément Java est un fichier, une classe, un champ, une methode, etc. Une ressource peut être un fichier XML, de propriétés, de *META-INF*, etc. Ces définitions sont tirées de Zimmermann et al[26].

Significativement pertinent Une entité significativement pertinente est une entité qui doit être changée pour accomplir la tâche de maintenance.

Hautement pertinent Une entité hautement pertinente est une entité qui permet de comprendre pourquoi et comment une entité significativement pertinente doit être changée.

Lee et Kang[37] ont, en plus, 2 autres types de pertinence mais nous ne les étudierons pas en détail ici.

1.2 But du mémoire

Un des buts de ce mémoire est d'essayer de voir comment une tâche de maintenance logicielle pourrait aider à donner des informations de compréhension logicielle pour la prochaine étape de maintenance. En effet, nous verrons au chapitre 2 qu'il existe de nombreux artéfacts logiciels qui contiennent des informations sur le système. La question est donc de déterminer comment nous pourrions utiliser une phase de maintenance pour en extraire certaines informations sur les fichiers.

Concrètement, ce mémoire vise à trouver un moyen de capturer le déroulement d'une tâche de maintenance et d'analyser les étapes effectuées par le programmeur pour finalement définir des fichiers significativement et hautement pertinents. De plus, ce mémoire tend à dire que les maintenances passées peuvent servir à la compréhension du logiciel pour la maintenance actuelle.

Pour ce faire, nous avons effectué une expérimentation sur des programmeurs devant effectuer des tâches de maintenance et nous avons capturé leurs interactions. Nous avons analysé leurs interactions pour essayer de déceler des critères permettant de différencier un fichier pertinent d'un fichier non pertinent.

La figure 1.1 représente la mise en contexte de notre mémoire. Elle montre une itération de maintenance. Il pourrait être intéressant de fournir des fichiers pertinents avant l'itération, de faire celle-ci et de donner en sortie d'itération des fichiers pertinents. Les fichiers pertinents obtenus à la fin seront utilisés pour la prochaine itération de maintenance. Nous avons l'idée d'un processus de maintenance itératif. Les fichiers pertinents fournis ne serviront que pour un seul projet. Pour ce faire, on divise l'itération de maintenance en 3 étapes : la capture de traces, la résolution de la tâche et l'analyse des résultats.

Ce mémoire va donc étudier la faisabilité d'une telle approche. La capture de traces sera manuelle ainsi que l'analyse des résultats. La résolution de la tâche sera effectuée par l'ensemble des sujets.

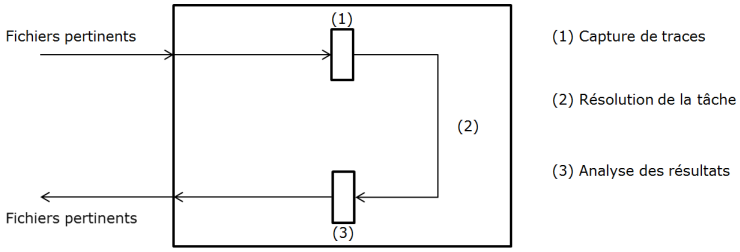


FIGURE 1.1 – Itération d'une maintenance

1.3 Plan du mémoire

Ce travail est divisé en deux parties.

La première partie présentera l'état de l'art lié à la compréhension logicielle et à la maintenance logicielle aux chapitres 2 et 3, respectivement. Le chapitre 2 dressera les points importants concernant la compréhension logicielle tandis que le chapitre 3 expliquera les différents points théoriques, principes et modèles de la maintenance logicielle.

La deuxième partie de ce travail présentera l'expérimentation. Chaque chapitre donnera de brefs rappels théoriques pour ensuite les mettre en contexte dans notre expérimentation. Le chapitre 4 donnera un bref aperçu du processus expérimental qui structurera la lecture des chapitres 5, 6, 7 et 8. Les chapitres 5 et 6 concerneront la phase préliminaire de l'expérimentation. Le chapitre 7 précisera la phase d'exécution de l'expérimentation. Le chapitre 8 donnera les résultats et leurs menaces.

Enfin, le dernier chapitre résumera les conclusions du travail et proposera des idées pour de futurs travaux dans ce domaine.

Première partie

Etat de l'art

Chapitre 2

Compréhension du logiciel

La compréhension du logiciel est une partie importante de la maintenance logicielle qui est expliquée au chapitre 3. Le logiciel doit être compris avant toute modification. De nombreuses recherches ont été effectuées pour essayer de diminuer le temps de compréhension d'un logiciel inconnu.

Dans ce chapitre, nous présenterons différentes techniques ou artéfacts qui peuvent mener à une meilleure compréhension d'un logiciel. Selon des textes scientifiques relevant ces techniques, il existe deux catégories de techniques se différenciant par la façon dont les propriétés d'un logiciel sont capturées[28]. Le premier groupement concerne l'ensemble des méthodes d'analyse statique, c'est à dire l'ensemble des méthodes extrayant des propriétés d'un logiciel sans devoir exécuter le programme. Le deuxième groupement concerne les méthodes d'analyse dynamique. Ces méthodes requièrent une ou plusieurs exécutions du programme pour obtenir des informations sur celui-ci.

Nous développerons ici certaines méthodes d'analyse statique et dynamique ainsi que les artefacts logiciels sur lesquels elles s'appliquent.

2.1 Méthodes d'analyse statique

Comme expliqué ci-dessus, les méthodes d'analyse statique d'un logiciel consistent à extraire de l'information depuis certains artéfacts d'un logiciel sans même exécuter celui-ci. Ces informations peuvent être extraites depuis plusieurs sources. Nous distinguerons ici 3 types de sources : les documents d'analyse conceptuelle et logique, le code source ainsi que les données externes non-structurées. L'ensemble de ces notions seront expliquées dans leurs sections respectives.

2.1.1 Analyse conceptuelle et logique

Ces documents sont l'ensemble des documents qui représentent les exigences du logiciel. Ils se focalisent uniquement sur ce que fait le logiciel et non sur la manière dont il doit le faire. Il est impératif, au niveau conceptuel, de s'abstraire de toute implémentation. Dans cette section, nous verrons différents moyens de représenter les exigences d'un logiciel par l'analyse des diagrammes de cas d'utilisations, l'analyse des flux de données, l'analyse des diagrammes entités-relations.

Pour l'ensemble des modèles, il existe peu de techniques automatiques d'aide à la compréhension. Il est donc important, pour pouvoir gagner de l'information de ces modèles, de comprendre leur fonctionnement et la façon dont ils sont construits. Ce mémoire ne visant pas ce niveau, nous nous limiterons dans les sections ci dessous à expliquer brièvement ces trois modèles de représentation conceptuelle et/ou logique d'un logiciel.

Diagrammes de cas d'utilisation

Les diagrammes de cas d'utilisation sont composés de trois éléments : les acteurs, les cas d'utilisation et les relations entre les cas d'utilisation.

Acteurs Les acteurs sont ceux qui vont avoir une interaction avec le système. Ils doivent être externes au système. Ceux-ci sont décrits par leurs rôles et capacités à interagir avec le système. Les acteurs sont soit des personnes soit d'autres systèmes qui interagissent dans le système que l'on décrit avec le diagramme de cas d'utilisation. En UML [1], un acteur est représenté par un symbole humanoïde.

FIGURE 2.1 – Acteur en UML

Les cas d'utilisation Un cas d'utilisation représente une fonctionnalité demandée au système. Celui-ci va représenter l'ensemble des scénarios d'interaction entre le système et le ou les acteurs concernés par la fonctionnalité dans un langage narratif. Les scénarios sont décrits en 2 parties. Ces 2 parties sont le scénario de base et les flux alternatifs. Les flux alternatifs ne sont pas toujours nécessaires mais représentent

1. Universal Modelling Language. Langage développé par l'*OMG*

des interactions qui ne suivent pas le cas normal d'exécution de la fonctionnalité. Par exemple, nous pourrions décrire un flux alternatif si l'interaction est annulée par un acteur. En UML, un cas d'utilisation est représenté par un ovale qui contient le nom du cas d'utilisation.

FIGURE 2.2 – Cas d'utilisation en UML

Les relations entre cas d'utilisation Les relations entre cas d'utilisation sont de 3 types : les relations simples, les *includes* et les *extends*.En UML, les relations simples sont représentées par des flèches au trait continu et les *inlcudes* et les *extends* par des flèches discontinues surmontées de la notation *includes* et *extends* respectivement.

Les diagrammes de cas d'utilisation permettent de rapidement comprendre l'ensemble des fonctionnalités demandées ainsi que les différents acteurs qui jouent un rôle dans le système[13, 33].

Diagrammes de classes

Les diagrammes de classes spécifient l'ensemble des propriétés statiques de chaque élément d'un domaine d'application.

Chaque classe du diagramme est spécifiée selon des attributs et des opérations. Une classe représente un concept du domaine d'application et non une instance réelle. Les attributs sont définis par leur type. Ces derniers peuvent être un *string*, un *char* ou autre en fonction du langage de programmation utilisé. Les opérations sont définies par un nom, leurs arguments et le type de retour qui peut être *void*. Les opérations ne sont pas définies par leur implémentation. Chaque exigence représente une opération. L'ensemble des classes est mis en relation par des associations. Ces dernières relient et fournissent une structure aux concepts ainsi qu'une hiérarchie dans les concepts.

Le diagramme de classe permet une vue d'ensemble des fonctionnalités et concepts contenus dans le système[33].

Modèle entité-association

Le modèle entité-association représente le domaine d'application au regard des données qu'il manipule. Celui-ci s'abstrait de toute information

d'implémentation technique d'une base de données. Il permet de représenter le domaine d'application d'un système sous formes d'entités et de leurs relations entre elles. Chaque entité sera dotée d'un ensemble de propriétés qui sont appelées attributs[31, 21].

2.1.2 Analyse du code source

Les documents de conception étant très rarement maintenus et accessibles, le code source est l'endroit où les programmeurs cherchent à comprendre le système. Ici, on entend par code source l'ensemble des données structurées par un langage de programmation qui constituent les spécifications exécutables du comportement d'un système. De plus, les systèmes devenant de plus en plus grands, il est difficile de concevoir qu'un nouveau développeur sur le projet puisse comprendre l'ensemble des documents de conception avant d'effectuer une tâche sur le système. Nous allons donc présenter les méthodes d'analyse statique du code source qui permettent une meilleure compréhension du logiciel.

La méthode d'analyse statique la plus couramment utilisée est l'utilisation des métriques. D'ailleurs, un nombre important de celles-ci existe. C'est pourquoi nous nous limiterons à présenter les classes majeures de métriques statiques ainsi qu'à en donner quelques exemples.Cette section est basée sur la thèse de doctorat de Caserta effectuée en 2012[30]. Si celles-ci sont généralement utilisées pour mesurer la qualité d'un logiciel, elles permettent aussi d'extraire de l'information permettant la compréhension d'un logiciel.

Les métriques de taille des composants Ces métriques sont les plus simples à calculer. Elles donnent rapidement une idée de la taille d'un composant par rapport à un autre ou par rapport au système complet. On peut donc rapidement voir où sont les composants plus importants par la taille. Les métriques de taille les plus souvent utilisées sont les *Lines of code(LOC)* et les *Number of classes(NOC)*. Les métriques de LOC calculent le nombre de lignes de code que contient un composant tandis que celles de NOC calculent le nombre de classes que contient un composant.

Les métriques de complexité des composants Ces métriques permettent de juger la complexité d'un composant logiciel. Les métriques de complexité sont utilisées dans la compréhension logicielle ainsi que dans la mesure de qualité d'un logiciel. En effet, on peut considérer qu'un composant complexe sera difficile à maintenir et à comprendre. La métrique la plus utilisée car indépendante du langage de programmation est le nombre *cyclomatique* de Mc Cabe. Cette métrique calcule le nombre de branches ou chemins qu'il est possible de prendre lors de l'exécution d'une méthode.

Les métriques de couplage entre les composants Les métriques de couplage permettent d'identifier ou de calculer les dépendances entre les composants. Plus les composants sont couplés, plus ils sont difficiles à comprendre. Bien qu'un bon nombre de métriques ait été trouvées, la métrique la plus utilisée est le *Couplage entre objets(Coupling Between Objects)* qui va calculer le nombre de classes couplées à la classe mesurée.

Les métriques de cohésion entre les composants Ces métriques calculent le nombre de connexions existantes dans une même classe. Elles vont donc mesurer la proportion des attributs utilisés par les méthodes de la classe par rapport à ceux qui sont instanciés par des méthodes ne venant pas de cette même classe. Ceci calcule la capacité d'une classe à instancier ses attributs. La métrique de complexité la plus utilisée est celle du *Manque de cohésion sur les méthodes(Lack of Cohesion On Methods)*.

Les métriques d'héritage Ces métriques calculent l'importance de l'héritage dans un système. Deux métriques sont couramment utilisées. Celles-ci sont la *profondeur de l'arbre de l'héritage(Depth in Inheritance Tree)* et le *nombre d'enfants d'une classe(Number of Children of a Class)*. Les métriques d'héritage sont utilisées pour juger de la complexité et de l'intensité d'utilisation de classes héritées dans un système.

L'ensemble des métriques de base présentées ci-dessus permettent de comprendre une partie de la conception et de l'organisation d'implémentation du système. Cependant, les nombreux outils utilisent des combinaisons de celles-ci ou des conditions logiques entre elles pour permettre une meilleure visualisation. Par exemple, on peut dire qu'une classe dont la complexité est forte et la cohésion est basse sera plus difficilement compréhensible et maintenable qu'une classe dont la complexité est faible et la cohésion forte.

2.1.3 Analyses des données externes non-structurées

Les recherches se portent de plus en plus sur les éléments extérieurs du logiciel pour aider à la compréhension et à la maintenance. Nous résumerons dans cette section les artéfacts qui contiennent des données non-structurées et ensuite les techniques qui permettent d'obtenir des informations sur celles-ci[38]. Nous reprendrons pour cela la définition de données non-structurées formulée par Manning : "Les données non-structurées sont les données qui n'ont pas une structure claire, sémantiquement déclarée, facile à comprendre par un ordinateur"[8]. Les données non-structurées correspondent donc aux données écrites dans un langage naturel.

Données non-structurées

Ces données peuvent être comprises dans différents endroits du logiciel ou dans l'environnement de développement : le code source, les bases de données de bugs, les chaines d'email ainsi que les systèmes de contrôles de versions.

Code source Le code source comprend l'ensemble des commentaires, noms de variables, méthodes ainsi que tous les littéraux compris dans les commandes d'impression vers une source externe comme la console ou des fichiers de logs.

Base de données de bugs Dans les bases de données de bugs telle que Bugzilla [2], nous pouvons obtenir de l'information à partir des *bug reports* [3].

Chaines d'email Dans les chaines d'email ainsi que dans les enregistrements des messageries instantanées fournies par certains logiciels de développement, les programmeurs laissent des informations pertinentes sur un système.

Système de contrôle de version Les systèmes de contrôle de version permettent d'inscrire un message lors de la publication d'une nouvelle version.

Techniques d'extraction des données non-structurées

Nous présenterons ici les étapes générales de pré-traitement de l'information et ensuite les différents systèmes de récupération d'information[38].

Pré-découpage La première phase d'extraction de données est le prétraitement de celles-ci. Elle consiste à enlever l'ensemble des bruits et données parasites pour obtenir un document contenant un maximum de données pertinentes. Cette phase comprend la *tokenization*, l'*identifier splitting*, le *stop word removal*, le *stemming* et le *pruning*.

- La *tokenization* consiste à casser le texte en mots, appelés tokens, tout en enlevant l'ensemble de la ponctuation ainsi que les caractères numériques.
- L'*indentifier splitting* consiste à diviser les noms de variables ou méthodes en utilisant certaines techniques telles que le *camel case* [4].

2. `http://www.bugzilla.org`

3. Un bug report est un message qui est posté dans une base de données de bugs et il permet d'avertir les autres développeurs d'un bug présent, avec une description, dans le système. De plus, des commentaires et patchs sont attachés aux bug reports

4. Le camel case est une technique écrivant les noms des méthodes et variables en montrant la séparation entre plusieurs mots par une majuscule. Un exemple de camel case est la fonction "ajout fonctionnalité" qui devient, avec cette technique, "ajoutFonctionnalité".

- Le *stop word removal* consiste à retirer les mots de liaisons ne transportant aucune information [5].
- Le *stemming* consiste à rechercher la racine des mots avec pour effet de supprimer les bruits liés à la conjugaison d'un verbe ou au pluriel d'un mot.
- Le *pruning* consiste à enlever l'ensemble des mots qui apparaissent trop souvent ou qui n'apparaissent pas assez souvent. En effet, ces mots pourraient apporter du bruit[35].

Systèmes de récupération de l'information D'après la définition de Manning[8], la récupération d'information permet de trouver parmi une large collection de documents, un document précis selon notre besoin. Il existe de nombreux systèmes qui permettent de récupérer de l'information. Nous allons présenter ici les 3 principaux évoqués que sont le *Vector Space Model*, le *Latent Semantic Indexing* et le *Latent Dirichlet Allocation*[38].

- le *Vector Space Model* est un modèle algébrique qui se base sur une matrice M termes sur N documents. Chaque entrée de la matrice représente le poids de ce terme dans le document. Pour calculer la similarité entre deux documents, il faut se baser sur les vecteurs colonnes. Pour éviter l'influence des termes trop communs, un système de pondération permet de minimiser ceux-ci[11].
- le *Latent Semantic Indexing* est une extension du *Vector Space Model*. En effet, il réduit la dimension de la matrice Termes/Documents d'un certain facteur K en regroupant des termes similaires en K sujet. Ceci permet de supprimer le bruit causé par la synonymie et la polysémie très courantes en langage naturel[36].
- Le *Latent Dirichlet Allocation* est un modèle probabiliste se basant sur une approche différente des deux autres modèles présentés ci-dessus. Il fait l'hypothèse que le corpus [6] contient un certain nombre de sujets et que dans chaque sujet, un terme a une certaine probabilité d'être généré. Il a l'avantage de privilégier la sémantique plutôt que la syntaxe[9].

2.2 Les méthodes d'analyse dynamique

L'analyse dynamique d'un logiciel, signifie en opposition à l'analyse statique, l'analyse du programme en exécution. Cela permet d'obtenir un ensemble d'informations générées lors de l'exécution d'un programme. Cette analyse permet aussi de se focaliser sur des scénarios spécifiques d'utilisation.

5. En français, les mots comme "et" et "ils" n'ont aucune information.
6. Le corpus est l'ensemble des documents. Voir [36].

Ce type d'analyse a été largement documenté depuis un certain nombre d'années. C'est pourquoi une étude fut menée pour tirer une vue d'ensemble du domaine[3].Le but du mémoire ne visant pas cette matière nous nous limiterons à donner ici les conclusions de cette étude.

Pour caractériser les différents articles qu'ils ont regroupés entre 1999 et 2008 dans cette étude, les auteurs[3] ont défini quatre facettes : l'activité, la cible, la méthode et l'évaluation.

Activité L'activité décrit ce qui est fait dans l'article

Cible La cible représente le type de logiciel ciblé par l'approche de l'article.

Méthode La méthode est la méthode d'analyse dynamique utilisée pour effectuer l'activité

Évaluation L'évaluation est la manière dont l'approche est validée.

Les auteurs ont ensuite défini pour chaque facette un ensemble d'attributs et affecté chaque article représentatif pour l'étude [7] d'un attribut par facette. Ceci a permis de connaitre les sujets les plus visés par la littérature.

Après regroupement de certains attributs, les auteurs ont calculé le nombre d'articles par attribut et par facette. Ci-dessous, se trouve l'attribut trié par facette dont le nombre d'articles en relation avec celui-ci est le plus grand.

Activité L'activité principale des documents scientifiques est de produire des vues d'ensemble de propriétés dynamiques.

Cible La plupart des systèmes ciblés sont des systèmes orientés objets.

Méthode La plupart des articles utilisent comme méthode des techniques de visualisation.

Évaluation En ce qui concerne l'évaluation, la plupart des articles étaient validés par l'utilisation de cas d'études open-source.

7. L'ensemble de l'étude a été réduite à l'étude de 172 articles

Chapitre 3

Maintenance logicielle

La maintenance logicielle [1] est l'une des étapes fondamentales dans le cycle de vie d'un système informatique.

Nous discuterons d'abord de la vision que nous avons de la maintenance et de ce que nous cherchons à faire avec celle-ci. Ensuite, nous évoquerons les différents types possibles de maintenance ainsi que les processus qui peuvent être mis en place pour les effectuer. Puis, nous parlerons des coûts engendrés par une maintenance. Enfin, nous verrons les modèles qui peuvent s'appliquer aux maintenances.

3.1 Vision d'une maintenance

Chaque maintenance est vue comme une entreprise temporaire initiée dans le but de créer un produit, un service ou un résultat unique[15]. Une maintenance est aussi un processus unique, qui consiste en un ensemble d'activités coordonnées et maitrisées comportant des dates de début et de fin, entrepris dans le but d'atteindre un objectif conforme à des exigences spécifiques telles que des contraintes de délais, de coûts et de ressources[17]. Les différentes maintenances d'un système sont donc vues comme des projets à part entière[24]. Cela nous conduit à dire que chaque maintenance est un projet unique où nous prenons en compte les caractéristiques propres et où nous adaptons la conduite du projet à chaque cas. De plus, une maintenance n'est ni une activité répétitive ni une mission permanente car elle a une date de début et de fin déterminée. Ceci la rend temporaire mais avec un résultat durable[24].

En outre, les maintenances incluent un ensemble très large d'activités comme la correction d'erreurs, l'ajout et la suppression de fonctionnalités et

1. La maintenance logicielle est appelée, dans la suite de ce travail, maintenance

l'optimisation du système tant au niveau de la vitesse d'exécution que de l'architecture du système en passant par l'amélioration de la maintenabilité du logiciel[23].

De plus, le pourcentage de développeurs qui doivent effectuer des opérations de maintenance est en constante augmentation. En effet, comme nous pouvons le voir ci-dessous, en 1950, 1 développeur sur 10 effectuait des maintenances tandis qu'en 2010, 65% des développeurs travaillaient sur la maintenance et l'évolution de système. De plus, les analystes prévoient qu'en 2020, 70% des développeurs travailleront sur des projets de maintenance[32] comme le montre la table 3.1.

Year	New projects	Enhancements	Repairs	Total
1950	90	3	7	100
1960	8,500	500	1,000	10,000
1970	65,000	15,000	20,000	100,000
1980	1,200,000	600,000	200,000	2,000,000
1990	3,000,000	3,000,000	1,000,000	7,000,000
2000	4,000,000	4,500,000	1,500,000	10,000,000
2010	5,000,000	7,000,000	2,000,000	14,000,000
2020	7,000,000	11,000,000	3,000,000	21,000,000

TABLE 3.1 – Distribution du travail des développeurs

3.2 But d'une maintenance

Une maintenance est effectuée pour suivre les besoins, en constante évolution, des systèmes d'information. Ces besoins d'évolution peuvent être tout autant des changements dans les exigences du système que dans l'environnement d'exécution dudit système[29].

Une maintenance est due à 3 éléments que sont la correction d'erreurs, le *business pull* et l'*IT push*[32].

Correction d'erreur La correction d'erreurs dans une maintenance consiste en la découverte d'un bug et en la résolution de celui-ci.

Business pull Le *business pull* réside dans tous les changements de haut-niveau qui peuvent modifier le comportement d'un système. Ces changements peuvent être une modification dans une loi ou dans les exigences de l'entreprise, ce qui fait que le logiciel doit évoluer pour respecter les nouvelles lois ou exigences lui permettant ainsi de continuer à survivre. Mais le

business pull ne se résume pas qu'à ces changements. En effet, une maintenance doit être effectuée lors d'une réorganisation, d'une fusion ou d'une acquisition d'entreprises pour que l'organisation résultante de ces changements puisse travailler avec le système. Une dernière partie du *business pull* est la création et suppression d'un produit. Les besoins des utilisateurs font également partie du *business pull*.

IT push L'*IT push* consiste en l'avènement de nouvelles technologies informatiques qui vont changer la relation avec le système. Nous pouvons citer l'apparition d'Internet, des appareils mobiles mais aussi de nouveaux systèmes d'exploitation et de nouveaux matériels.

3.3 Types de maintenance

Il existe 3 types principaux de maintenance pour un système informatique. Ce sont les maintenances adaptatives, correctives et perfectives[4, 27, 32]. Une évolution logicielle est perçue comme une maintenance adaptative et perfective. Il existe deux autres types de maintenance que sont la maintenance préventive[23, 1] et la maintenance d'urgence[16]. Ces types sont catégorisés en fonction du but de la maintenance.

Adaptative La maintenance adaptative est effectuée suite à un changement dans l'environnement ou dans les exigences.

Corrective La maintenance corrective est effectuée suite à la découverte d'une erreur qui n'a pas été détectée à la livraison du produit. Les erreurs peuvent survenir à cause d'une mauvaise conception du design, une logique métier incorrecte ou des erreurs de codage[1].

Perfective La maintenance perfective est effectuée suite à un changement des besoins des consommateurs. Cette maintenance est vue comme une amélioration fonctionnelle et non-fonctionnelle du système pour que celui-ci tende vers la perfection[1]. En effet, le succès d'un logiciel n'est pas fait dès sa première itération mais bien par les itérations suivantes où le logiciel s'est adapté aux besoins des consommateurs et que celui-ci soit devenu parfait pour eux.

Préventive La maintenance préventive est effectuée pour pallier la menace potentielle d'une erreur mais aussi pour faciliter la maintenance et la compréhension du système[23] car les 3 premiers types de maintenance ajoutent de la complexité. Cette maintenance est vue comme du *refactoring* de code quand le système fonctionne correctement. Ce type de maintenance

s'effectue aussi pour avoir une documentation du système qui reste cohérente avec le code qu'elle explique.

D'urgence La maintenance d'urgence est une opération non prévue qui permet de garder le système opérationnel alors qu'une maintenance corrective est déjà en cours. Ce type de maintenance fait partie des maintenances correctives[16].

Après avoir détaillé ces 5 types de maintenance, nous pouvons observer qu'une maintenance *traditionnelle* ne reprend que la maintenance corrective car les autres sont vues comme une évolution du logiciel[32, 27]. La maintenance d'urgence est bien entendu une maintenance *traditionnelle* mais faisant partie des maintenances correctives, elle n'est pas différente de celles-ci.

Ces types de maintenance n'ont pas le même pourcentage de répartition durant le cycle de vie d'un système. Leurs différents pourcentages restent sensiblement identiques selon les auteurs.

Comme nous pouvons le voir sur la figure 3.1, Klint[32], qui ne distingue que 3 types de maintenance, estime qu'un peu plus de la moitié des maintenances est de type perfective et que le reste de celles-ci se partage de manière relativement équivalente entre les maintenances correctives et adaptatives. Cette répartition induit que 80% des maintenances sont vues comme des évolutions du logiciel.

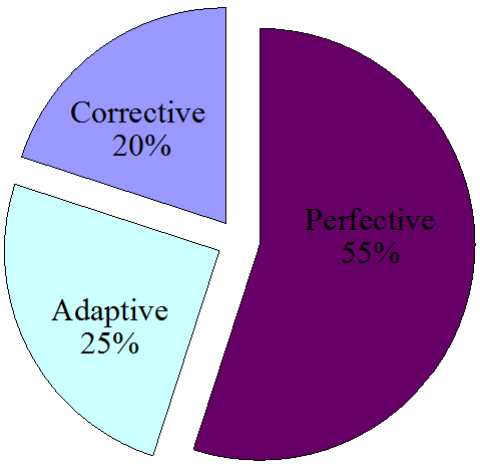


FIGURE 3.1 – Pourcentage de répartition des types des tâches de maintenance pour Klint

Pour Aggarwal et Senghi[23], malgré les maintenances préventives, les pourcentages de répartition sont quasiment équivalents comme nous pouvons le voir à la figure 3.2.

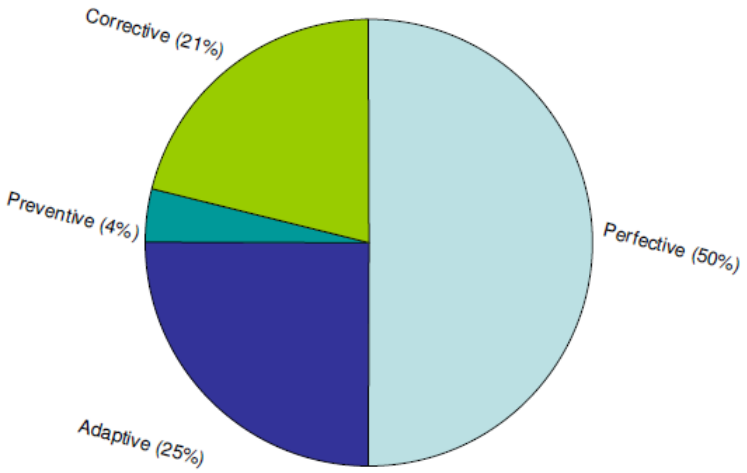


FIGURE 3.2 – Pourcentage de répartition des types des tâches de maintenance pour Arrgarwal et Singh

Il ne faut pas oublier que certains auteurs donnent des noms différents pour ces types de maintenance et que ces derniers peuvent avoir un sens différent d'un auteur à l'autre. Par exemple, Sommerville[14] cite 3 types de maintenance que sont la réparation d'erreurs, l'adaptation d'environnement et l'ajout de fonctionnalités. La réparation d'erreurs correspond à une maintenance corrective et nous sommes dans le cas d'un simple changement de noms. L'adaptation d'environnement est une adaptation du système pour un nouvel environnement alors qu'une maintenance adaptative est une adaptation à un nouvel environnement mais aussi à de nouvelles exigences. Nous sommes alors dans un changement de sens. De même qu'une maintenance perfective signifie l'ajout de fonctionnalités ou l'amélioration de la structure et des performances, une adaptation du système n'ajoute que des fonctionnalités à celui-ci et nous nous retrouvons dans le cas d'un changement de sens.

3.4 Étapes d'une maintenance

Lors d'une maintenance, différentes étapes sont réalisées pour que le système atteigne le résultat attendu. Ces étapes forment un processus de maintenance. Ce processus commence toujours par une requête de changement. Différents processus existent et ont un nombre d'étapes variables. Les pro-

cessus sont importants car ils donnent des guides pour dire ce qui doit être fait et comment cela doit être fait[7] [2].

3.4.1 Le processus selon l'IEEE

Le processus de maintenance proposé conjointement par l'*IEEE* et l'*ISO* explique les différentes étapes qui peuvent arriver entre la fin du développement initial du système et la mise à la retraite de ce dernier[16]. Il comporte 6 étapes montrées par le figure 3.3.

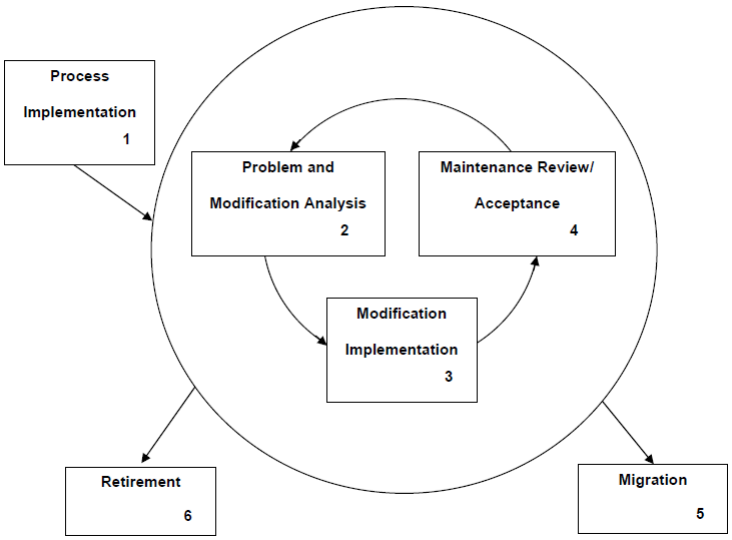


FIGURE 3.3 – Processus de maintenance IEEE

Les étapes 2, 3 et 4 sont celles qui seront exécutées plusieurs fois durant la vie du logiciel tandis que les étapes 1, 5 et 6 ne sont effectuées qu'une seule et unique fois sur le système.

Implémentation du processus Cette étape consiste à établir les plans et les procédures qui seront exécutés durant le processus de maintenance.

Analyse du problème et de la modification Cette étape est effectuée quand survient une requête pour modifier le système. Ce dernier est alors analysé pour détecter les modifications qui devront être réalisées. Ces modifications sont analysées à leur tour pour évaluer les conséquences qu'elles auront sur l'organisation et le système. Quand les conséquences sont jugées acceptables, les modifications sont approuvées pour être implémentées.

2. Les modèles définis au point 3.6 sont aussi importants et donnent les mêmes avantages.

Implémentation de la modification Cette étape consiste à implémenter les modifications et à les tester.

Acceptation de la maintenance Cette étape consiste à s'assurer que les modifications sont correctes et qu'elles ont été effectuées avec les standards appropriés ainsi qu'avec la bonne méthodologie.

Migration Cette étape consiste à documenter et à développer les étapes requises pour effectuer une migration vers un nouvel environnement. Cette migration doit être conforme avec la norme ISO 12207. Cela signifie qu'un plan de migration doit être créé. De plus, les utilisateurs doivent être formés en plus d'être avertis du début de la migration et de la fin de celle-ci. En outre, les impacts cette migration ainsi que toutes les données doivent être archivés.

Retraite Cette étape consiste en la fin de vie du système. Une analyse doit être effectuée pour aider à la décision de mettre, le cas échéant, le produit à la retraite. Cette analyse est la plupart du temps basée sur l'économie. De plus, des aspects comme l'utilité de garder de vieilles technologies, de passer à une nouvelle en changeant de projet ou de développer un nouveau système pour en avoir un plus modulaire, plus facile à maintenir, respectant un standard ou pour faciliter l'indépendance par rapport à un vendeur sont à prendre en compte avant de mettre un système à la retraite.

3.4.2 Le processus selon Aggarwal et Singh

Ce processus consiste en 5 étapes chronologiques : l'identification du problème, la compréhension, la proposition, la prise en compte des répercussions et les tests[23].

Identification du problème Cette étape consiste à identifier la source de la requête de changement et à savoir si c'est une maintenance pour l'ajout de fonctionnalités, pour la correction d'erreurs, pour la suppression de fonctionnalités obsolètes ou pour une optimisation.

Compréhension Cette étape consiste à analyser le système pour en saisir le fonctionnement et appréhender comment et où effectuer la maintenance. La compréhension logicielle passe par l'analyse des différents documents définis au chapitre 2 et par l'utilisation des techniques expliquées aussi au chapitre 2.

Proposition Cette étape consiste en l'implémentation d'une solution pour résoudre la requête qui a été définie à l'étape d'identification du problème.

Prise en compte des répercussions Cette étape consiste en la prise en compte de tous les effets secondaires que peut avoir, pour le système, l'implémentation de la solution. De plus, lors de cette étape, nous essayons de nous prémunir contre ces effets secondaires.

Tests Cette étape consiste à tester le système après l'implémentation de la solution et à vérifier que ce dernier se comporte correctement et garde au moins le même niveau de fiabilité qu'avant les changements. Si les tests sont négatifs, nous revenons à la première étape.

Ce processus est représenté par la figure 3.4.

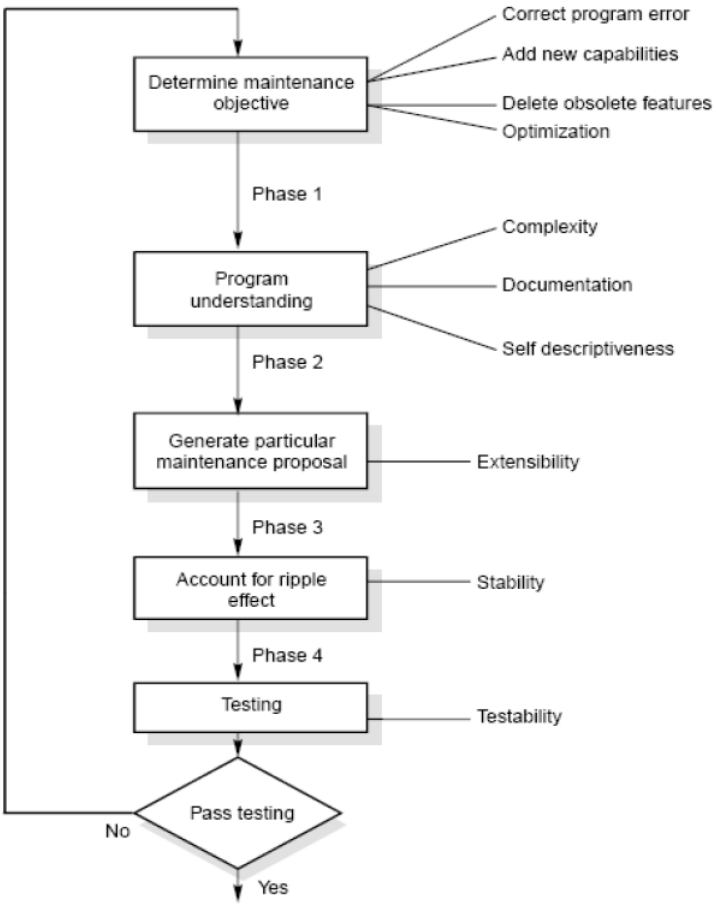


FIGURE 3.4 – Processus de maintenance logicielle

3.5 Coûts d'une maintenance

Quand nous parlons de maintenance, nous voyons les types de maintenance à faire, les changements à produire, les étapes à faire mais aussi les coûts associés aux modifications à effectuer. Ces coûts sont tant humains et financiers que temporels. De plus, le temps et l'argent sont corrélés de manière proportionnelle, ce qui fait que, lorsque le temps augmente, l'argent dépensé augmente également.

Bien que de nombreuses recherches aient été conduites pour diminuer les coûts financiers, ces derniers n'ont pu être réduits de manière significative[1] et restent compris, la plupart du temps, entre 60 et 80% de l'argent dépensé pendant la vie totale d'un système d'information. Cette dépense lors de la maintenance peut augmenter jusqu'à 90% du total des frais. Ce coût est compris entre 2 fois et 100 fois le prix du développement initial[32]. Par exemple, pour les systèmes en temps réel, le coût de maintenance est jusqu'à 4 fois supérieur au coût de développement initial[14].

Le coût étant si élevé, il serait intéressant de le réduire et comme le coût financier est lié au coût temporel, une constatation naïve serait de dire qu'il faut réduire le temps passé lors d'une maintenance pour réduire l'argent dépensé mais de nombreux problèmes surviennent pour comprendre le système et réduire le temps que les développeurs passent sur son étude. Les techniques expliquées au chapitre 2 peuvent aider à la compréhension du système. Ces problèmes sont entre autres, la structure et l'âge du système, le niveau technique et la stabilité de l'équipe de développement et les mauvaises pratiques de développement de cette dernière[14]. Les problèmes liés à la structure et l'âge du logiciel sont confirmés par les lois de Lehman[25]. En effet, la complexité augmente avec le temps tandis que la qualité diminue. Ceci qui rend la structure plus difficile à comprendre au fur et à mesure des changements. La stabilité et le niveau technique de l'équipe de développement influencent le temps de maintenance car plus l'équipe est changeante, moins celle-ci connait le système. De même, si le niveau technique de l'équipe est faible, les changements prendront plus de temps pour être effectués tandis que si l'équipe a un niveau technique élevé, les changements seront effectués rapidement. En outre, les mauvaises pratiques de développement peuvent influencer les opérations de maintenance car ces dernières ne sont pas forcément faites par l'équipe de développement et la précédente équipe de maintenance[14].

En outre, selon Aggarwal et Senghi[23],les coûts peuvent augmenter suite à diverses causes comme un manque d'informations sur le système, ce qui implique que les changements peuvent avoir été effectués par des personnes ne comprenant pas le code clairement. Cela a pour conséquence de réduire la qualité du système mais rend aussi la compréhension mal aisée pour effectuer les prochaines modifications. De plus, la documentation peut être erronée, ce qui ralentit le temps de compréhension pour effectuer la maintenance. De même, certains systèmes ne sont pas prévus pour évoluer, cela rend leur maintenance plus difficile.

Enfin, à moins d'avoir un système auto-adaptant, une maintenance met, généralement, le système hors-ligne. Ce dernier devient indisponible et pro-

voque pour l'entreprise un manque à gagner qui peut être élevé si la maintenance dure longtemps[12].

3.6 Modèles de maintenance

Pour effectuer des maintenances, il existe plusieurs modèles qui sont basés chacun sur une philosophie différente. Ceux-ci sont le modèle *quick-fix*, *amélioration itérative*, *orienté réutilisation*, *Boehm maintenance de Taute*, *Osborne* et *cycle de vie conscient de la maintenance*.

3.6.1 Modèle quick-fix

Le modèle *quick-fix* attend qu'un problème apparaisse pour le résoudre aussi rapidement que possible. Ce modèle modifie généralement juste le code source du système en accord avec les objectifs de maintenance. Il tire ses racines dans la méthodologie de développement en cascade[22]. Il est le plus approprié pour effectuer des maintenances correctives et adaptatives simples[22]. Souvent, ce modèle est utilisé comme une mesure temporaire pour résoudre un bug tandis que la vraie correction sera implémentée ultérieurement, en même temps que d'autres corrections et améliorations, dans une mise-à-jour majeure[1].

Les changements sont effectués sans planification, sans analyse d'impact et de design et sans test de régression pour réduire le temps entre la requête de maintenance et la fin de cette dernière[1, 10]. De plus, les entreprises utilisant ce type de maintenance croient que les utilisateurs courent un plus grand risque en les laissant attendre qu'en leur proposant la solution la plus rapide[10].

La faiblesse du modèle *Quick-fix* est que les modifications se font sous forme de patchs. Ces derniers ne sont pas bien documentés et peuvent détruire partiellement la structure du système, tout en freinant les futures évolutions[41, 5].

Le modèle *Quick-fix* est montré par la figure 3.5.

3.6.2 Modèle d'amélioration itérative

Le modèle d'amélioration itérative a été originellement proposé comme un modèle de développement[22]. Ce modèle commence avec les exigences, le design, le code, les tests et la documentation du système existant. Il est composé de 3 phases : l'analyse de l'existant, la caractérisation des modifications et l'implémentation des changements[1, 23] comme nous le montre la figure 3.6.

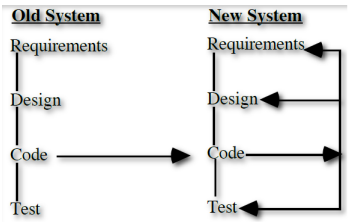


FIGURE 3.5 – Modèle de maintenance Quick-fix

Une analyse du système existant est mise en place pour regarder les modifications nécessaires afin de satisfaire les exigences définies par la requête de changement. Après cette analyse, le système est modifié en commençant par les documents de plus haut niveau affectés par les changements en descendant, ensuite, dans les niveaux inférieurs. Ces modifications sont propagées à travers l'ensemble des documents et permettent le re-design de chaque étape[1, 22]. Ces étapes sont : les exigences, le design, le codage, les tests et l'analyse de risques[1] comme nous pouvons le voir sur la figure 3.6.

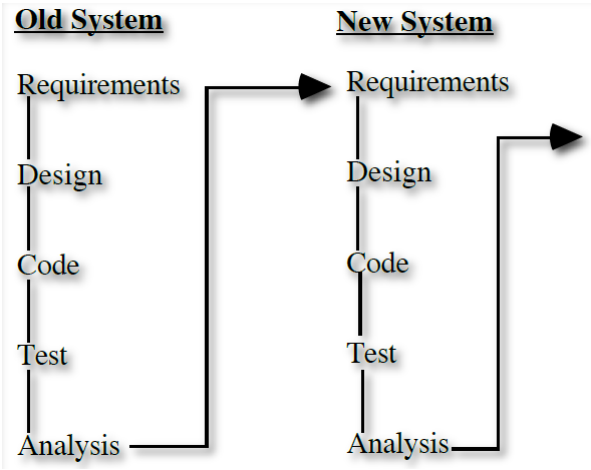


FIGURE 3.6 – Modèle de maintenance d'amélioration itérative

Ce modèle supporte explicitement la réutilisation et peut changer certains modèles comme le *quick-fix*[1]. Il est le plus approprié pour des maintenances perfectives où les exigences ne sont pas modifiées de manière profonde entre le système existant et le nouveau système[22].

Le problème de ce modèle provient des suppositions faites sur l'existence d'une documentation complète et de la faculté de l'équipe de maintenance à analyser complètement le système actuel[1].

3.6.3 Modèle orienté réutilisation

Le modèle orienté réutilisation comprend 4 étapes : l'identification, la compréhension, la modification et l'intégration[39, 1, 23, 41] comme nous le montre la figure 3.7. Les nouveaux composants sont construits *from scratch*[22].

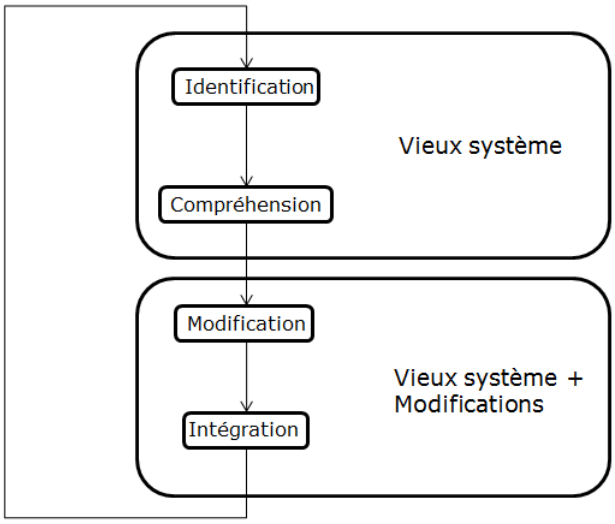


FIGURE 3.7 – Modèle de maintenance orienté réutilisation

Identification Cette étape identifie les parties du système qui peuvent être réutilisées dans le nouveau système.

Compréhension Cette étape correspond à la compréhension des parties qui peuvent être réutilisées.

Modification Cette étape vise à modifier les parties de l'ancien système pour que celles-ci soient en accord avec les nouvelles exigences.

Intégration Cette étape permet l'inclusion des nouvelles parties dans l'ancien système.

Ce modèle supporte un plus large champ de types de maintenance que les précédents. En effet, les 3 types principaux de maintenance sont supportés par ce modèle[22].

3.6.4 Modèle de Boehm

Le modèle de Boehm est présenté à la figure 3.8.

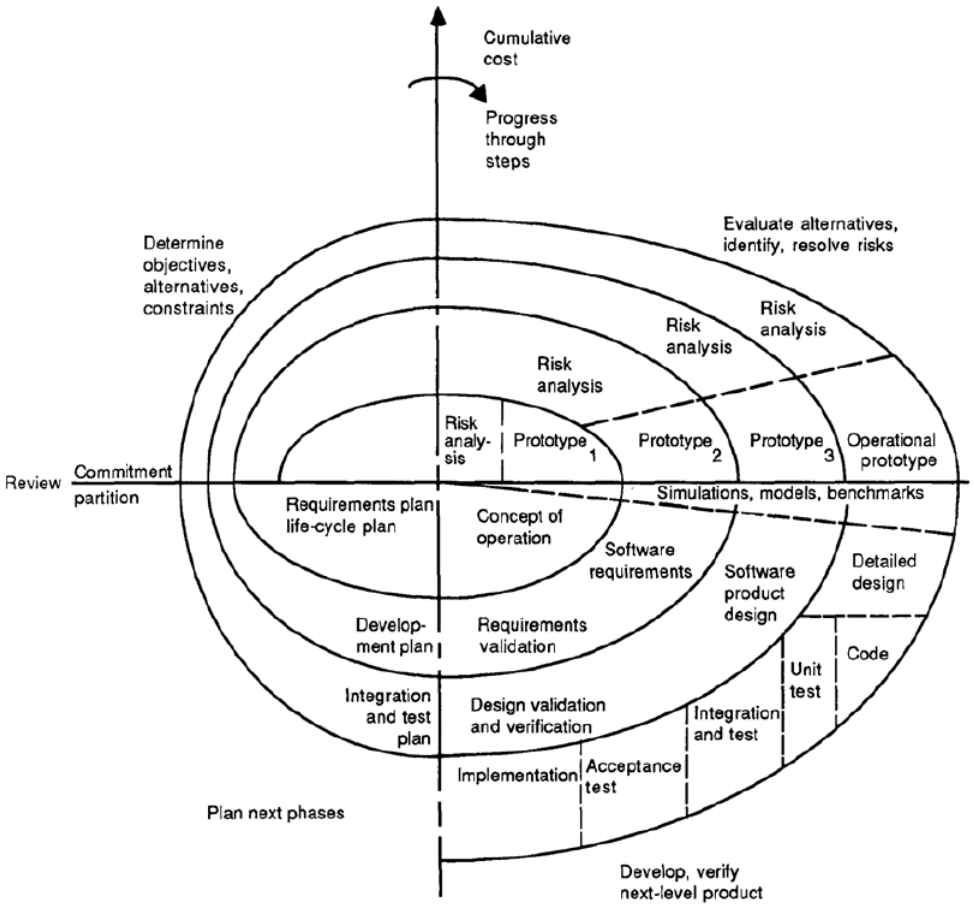


FIGURE 3.8 – Modèle de maintenance de Boehm

Ce modèle se base sur une boucle fermée utilisant les principes et modèles économiques. C'est-à-dire que le management est la force conductrice d'une maintenance. Il dicte les changements à effectuer en se basant sur une analyse coût/bénéfice[1]. Selon Boehm, il ne permet pas de juste augmenter la productivité mais également la compréhension du processus[1, 5]. D'un point de vue économique, ce modèle est très efficace mais il peut mettre le système en danger à cause d'un mauvais jugement, de décisions biaisées, etc car le système est mis en utilisation avant l'évaluation des changements[5, 34].

Le modèle de Boehm ne peut faire une estimation du temps de maintenance alors qu'une entreprise pourrait vouloir savoir le temps nécessaire pour incorporer les modifications demandées par la requête de changement[5, 34].

Ce modèle utilise une quantité appelée *Annual Change Traffic (ACT)*. Cette quantité est définie comme l'addition des lignes ajoutées, modifiées et

supprimées durant un an divisée par le nombre de lignes totales du projet[23].

$$ACT = \frac{KLOC_{added} + KLOC_{deleted}}{KLOC_{total}} \tag{3.1}$$

L'ACT est utilisé pour calculer l'*Annual Maintenance Effort (AME)*. L'AME est le résultat de la multiplication de l'ACT, de l'*effort de développement logiciel en personne mois (SDE)* et du *facteur d'ajustement d'effort (EAF)*[23].

$$AME = ACT \times SDE \times EAF \tag{3.2}$$

3.6.5 Modèle de Taute

Le modèle de Taute se base sur un cycle en 8 phases typiques montrées sur la figure 3.9.

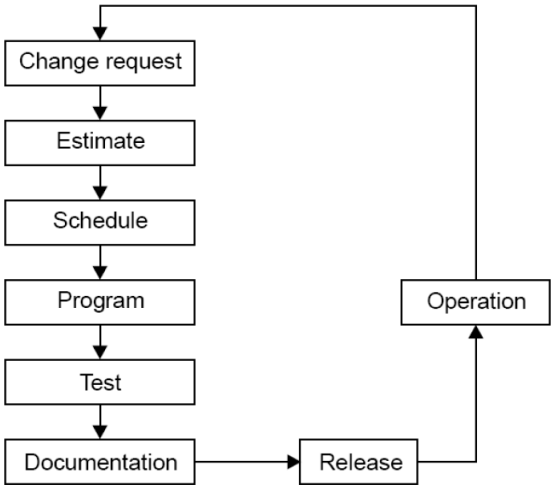


FIGURE 3.9 – Modèle de maintenance de Taute

Quand ce modèle est mis en place dans un processus de maintenance, les modifications sont effectuées sans faire d'analyse de faisabilité de la requête alors qu'elle est un point crucial dans chaque maintenance[34]. De plus, après l'estimation de l'effort et du facteur de temps, celle-ci n'est pas donnée au client pour acceptation alors que le client doit être au courant de ces paramètres pour implémenter les changements[34].

3.6.6 Modèle d'Osborne

Le modèle d'Osborne a la particularité de gérer directement la réalité au lieu de supposer un monde idéal avec, par exemple, une documentation complète du système[1].

Ce modèle est traité comme des itérations continues du cycle de vie du logiciel avec, à chaque étape, des prévisions faites pour améliorer la maintenance[42, 1, 10]. Le modèle de maintenance d'Osborne est présenté par la figure 3.10.

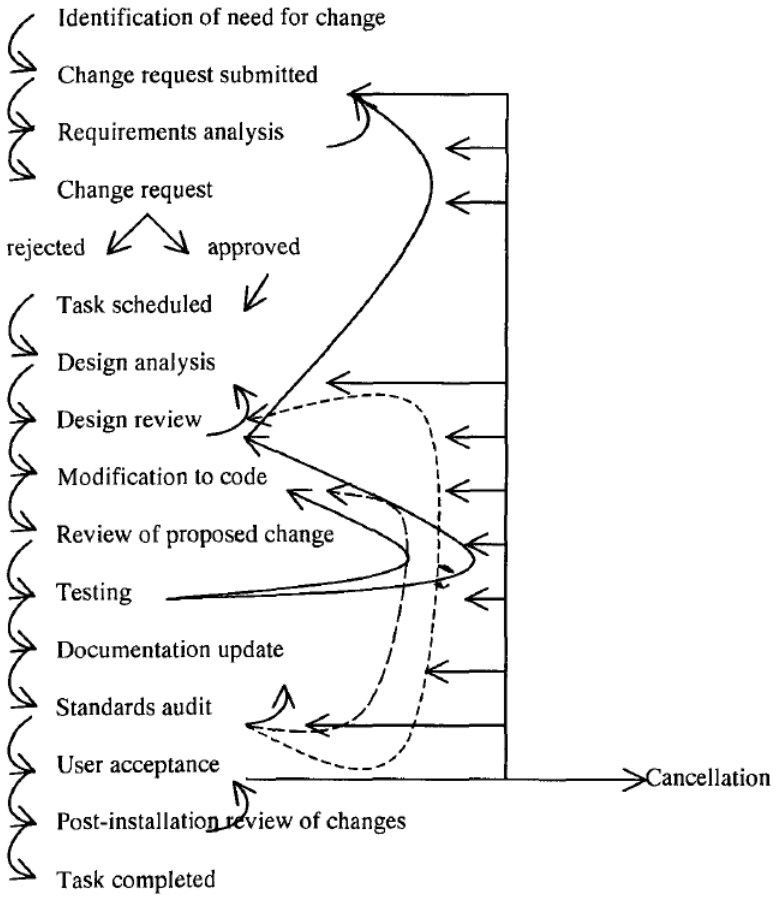


FIGURE 3.10 – Modèle de maintenance d'Osborne

Osborne suppose que les nombreux problèmes techniques survenant durant une maintenance sont dus à des communications et à un contrôle inadéquats du management[42, 1, 10]. Elle recommande une stratégie qui inclut les éléments suivants[1] :

- La prise en compte des exigences de la maintenance dans la spécification des changements.
- Un programme d'assurance qualité qui établit des exigences d'assurance qualité.
- Un moyen de vérifier que les objectifs de la maintenance ont été atteints.

– Une revue des performances pour fournir du *feedback* aux managers.

3.6.7 Modèle du cycle de vie conscient de la maintenance

Le modèle du cycle de vie conscient de la maintenance se base sur une boucle en 14 étapes qui sont montrées sur la figure 3.11.

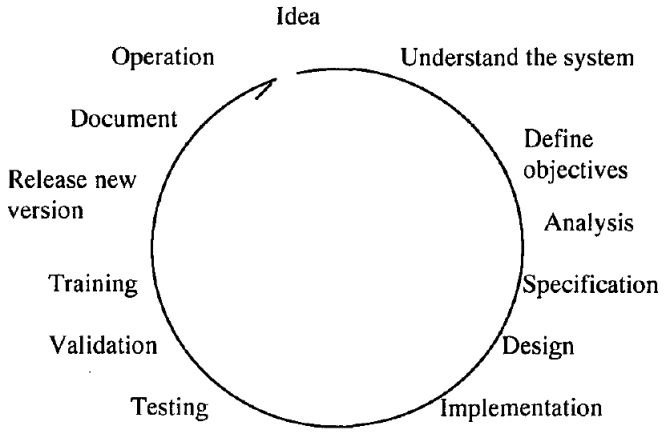


FIGURE 3.11 – Modèle du cycle de vie conscient de la maintenance

Ce modèle reconnait le besoin de la maintenance dès le début du développement du système. Cela a pour conséquence de fournir plus d'efforts au moment du développement initial du système par rapport aux modèles de développement traditionnels comme le modèle en cascade mais permet que moins de ressources soient mobilisées lors des phases de maintenance[1]. La figure 3.12 représente ce modèle.

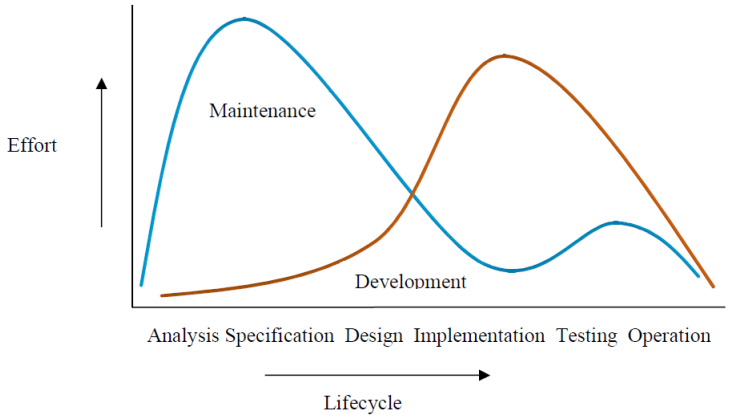


FIGURE 3.12 – Effort nécessaire lors des différentes étapes du système

Deuxième partie

Expérimentation

Chapitre 4

Aperçu du processus expérimental

Conduire une expérimentation n'est pas chose aisée. Nous devons la préparer correctement parce qu'une fois lancée, elle ne peut être modifiée. Cette préparation est l'un des avantages principaux lors de la conduite d'une expérimentation parce que les expérimentateurs peuvent contrôler certaines variables comme les sujets et l'environnement. Cette préparation permet aussi d'effectuer plus facilement une réplication de l'expérimentation et d'effectuer des analyses statistiques sur les résultats obtenus pour vérifier ou tester notre théorie ou hypothèse.

Une expérimentation permet de tester une relation entre un traitement et un résultat. Si l'expérimentation est mise en place correctement, nous devrions être capables de tirer des conclusions à propos de l'hypothèse qui a été émise. Pour être certains que nous puissions tirer des conclusions positives ou négatives d'une expérimentation, nous mettons en place un processus qui permet d'éviter certains problèmes. Ce processus est pour l'essentiel tiré de Wohlin[7] qui est une référence de base en la matière. Les premières sections des chapitres 5, 6, 7 et 8 sont des résumés théoriques de Wohlin[7].

4.1 Variables, traitements, objets et sujets

Avant de caractériser le processus d'une expérimentation, nous devons expliciter certains termes pour en établir une définition acceptée tant par nous que par le lecteur.

Variables Lors d'une expérimentation, il y a deux types de variables : soit les variables indépendantes, soit les variables dépendantes. Les variables indépendantes sont celles qui sont contrôlées tandis que les variables dépendantes sont les variables de réponses car elles permettent d'étudier l'effet de

changements dans les variables indépendantes. Il n'y a souvent qu'une seule variable dépendante. En d'autre mots, les variables indépendantes peuvent prendre au minimum 2 valeurs de variables susceptibles d'influencer d'autres variables, les variables dépendantes. De plus, les variables indépendantes ne dépendent pas du sujet observé au-cours de l'expérimentation.

La relation qui s'établit entre les variables indépendantes et dépendantes au-cours d'une expérimentation est montrée par la figure 4.1.

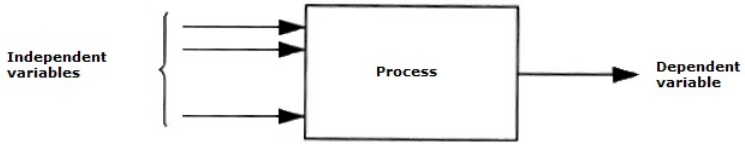


FIGURE 4.1 – Illustration des variables dépendantes et indépendantes

Facteurs Les facteurs sont les variables indépendantes. L'effet provoqué à la suite de leur changement est étudié lors de l'expérimentation. Les autres variables indépendantes sont fixées pour permettre d'analyser correctement les facteurs.

Traitement Un traitement est une valeur particulière d'un facteur. Le choix du traitement ainsi que le niveau des autres variables indépendantes font partie du design de l'expérimentation. Les traitements sont appliqués à un ensemble d'objets et de sujets.

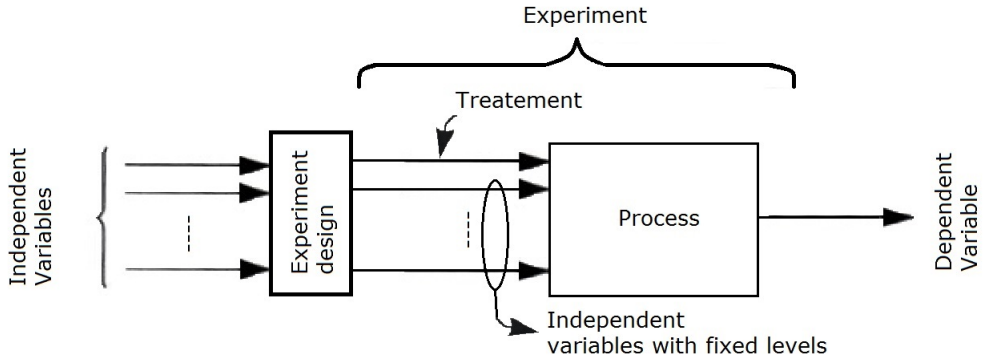


FIGURE 4.2 – Illustration d'une expérimentation

La figure 4.2 montre effectivement que le design de l'expérimentation fixe les différents niveaux pour les variables indépendantes mais choisit aussi les différents traitements pour les facteurs.

Objets Un objet peut être, par exemple, un document qui doit être revu pour fixer un bug ou de la documentation à mettre à jour, etc.

Sujets Les sujets sont les personnes qui appliquent le traitement.

Tests Une expérimentation consiste en une batterie de tests où chacun résulte de la combinaison de traitements, de sujets et d'objets. Le nombre de tests nous fournit une idée de l'effet d'un facteur pour l'expérimentation.

4.2 Processus expérimental

Après avoir défini les termes dont nous allons user dans la suite de ce mémoire, nous pouvons enfin passer à la définition du processus expérimental que nous allons exploiter. Comme nous l'avons déjà dit au chapitre 3, un processus permet de donner des guides pour savoir quoi faire et à quel moment tout en disant comment le faire. Cela minimise les erreurs.

Le processus expérimental peut être défini en 5 étapes : la définition, la planification, l'opération, l'analyse et l'interprétation, la présentation et le package.

Comme nous le montre la figure 4.3, nous pourrions penser que ce processus s'applique selon un modèle en cascade où chaque étape doit être finie avant de pouvoir passer à la suivante alors que cela n'est défini nulle part. En effet, l'ordre est purement indicatif et rien ne dit que nous ne pouvons revenir en arrière pour modifier les étapes précédentes. Le processus est donc vu comme un modèle itératif[7] jusqu'à un certain point. Lorsque la phase d'opération a commencé, il est impossible de revenir en arrière sans courir le risque d'être dans l'impossibilité de réutiliser les mêmes sujets car ceux-ci sont influencés par l'expérimentation et peuvent avoir gagné des connaissances entretemps.

La suite de ce mémoire se structure selon ce processus. En effet, nous commencerons par la définition de l'expérimentation dans le chapitre 5 qui sera suivie par la planification de cette expérimentation au chapitre 6. Nous passerons ensuite à la phase opérationnelle dans le chapitre 7 pour terminer par l'analyse et l'interprétation des données et résultats dans le chapitre 8. Le présent mémoire constitue la phase de présentation de notre expérimentation.

Ces 4 chapitres s'organisent en 2 parties. La première partie portera sur un rappel théorique concernant la phase en cours de processus expérimental

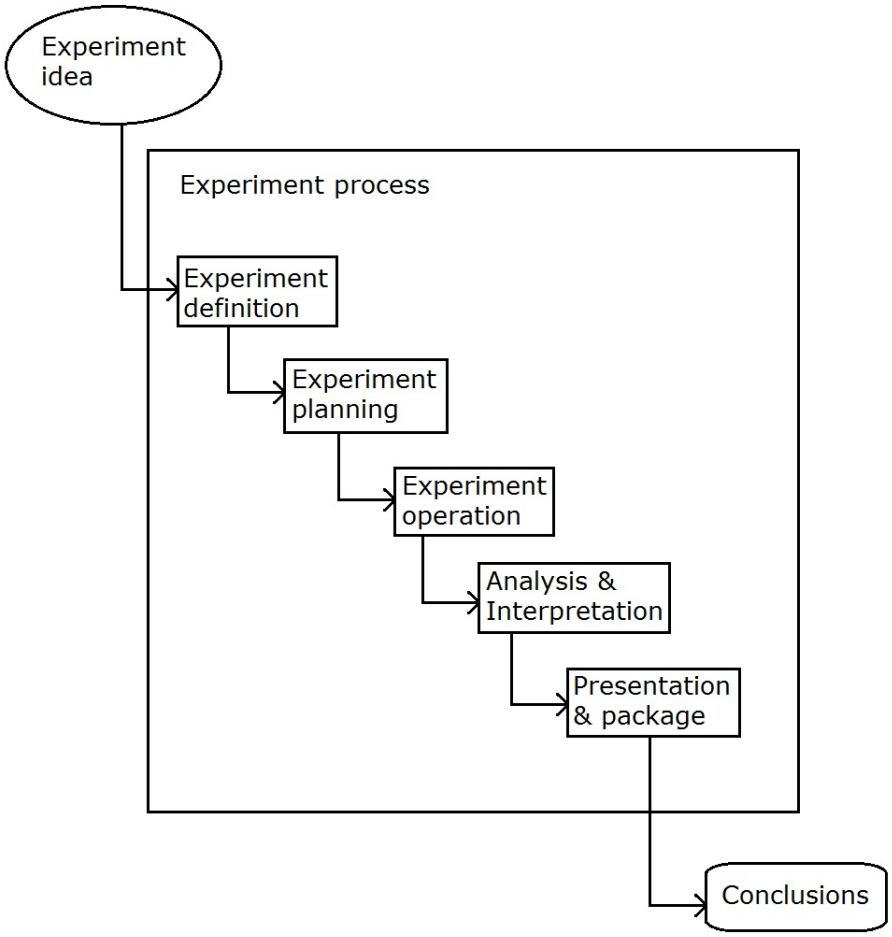


FIGURE 4.3 – Processus d'une expérimentation

tandis que la deuxième partie présentera cette même phase mais adaptée à notre cas.

Chapitre 5

Définition de l'expérimentation

Ce chapitre va définir notre expérimentation en exprimant les objectifs que nous avons voulu atteindre à partir d'un constat : les développeurs passent trop de temps, lors des étapes de maintenance, à comprendre le code au lieu de le modifier pour résoudre la tâche qui leur a été confiée. De plus, les développeurs passent en moyenne 35% de leur temps dans les mécanismes de navigation entre fichiers[2].

Nous passons par des rappels théoriques à la section 5.1 qui montreront comment nous avons pu définir notre expérimentation à la section 5.2. Les rappels théoriques se basent sur Wohlin[7].

5.1 Phase de définition d'une expérimentation : rappels théoriques

La présente section résume brièvement la théorie associée à cette phase de définition d'une expérimentation. Nous expliquons pourquoi cette phase est importante, le modèle qui est à notre disposition pour réduire les risques d'une mauvaise définition et la concrétisation de l'idée de cette expérimentation.

Cette étape d'un processus expérimental est fondamentale car elle jette les bases de l'expérimentation. Si ces bases ne sont pas bien définies, une nouvelle définition de l'expérimentation devra être faite ou pire, l'expérimentation pourrait ne pas être utilisable. La définition d'une expérimentation passe par la transformation d'une idée en une définition concrète comme nous le montre la figure 5.1. L'idée de l'expérimentation est expliquée à la

section 5.2.1 tandis que l'expérimentation elle-même est définie à la section 5.2.2.

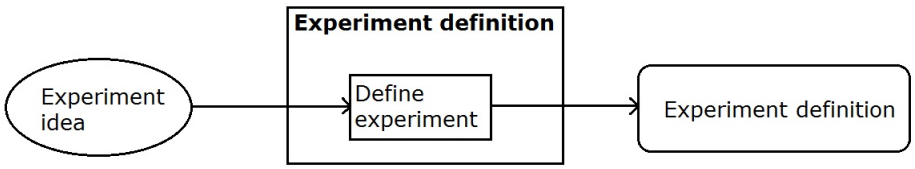


FIGURE 5.1 – Phases de la définition d'une expérimentation

Pour minimiser les risques, nous suivons le modèle GQM[1][7]. Ce modèle nous permet d'assurer que les points importants d'une expérimentation soient clairement fixés avant le commencement des autres phases. Ce modèle est défini comme-ceci :

Analyser <Objet(s) de l'étude>
dans le but de <But>
en fonction de leurs <Propriétés>
du point de vue de <Perspective>
dans le contexte de <Contexte>

Objet(s) de l'étude L'objet de l'étude est l'artéfact qui est observé durant l'expérimentation. Un objet peut être un produit, un processus, une ressource, un modèle, une métrique ou une théorie.

But Le but définit l'intention de l'expérimentation ; ce que l'on veut prouver.

Propriétés Les propriétés sont les effets primaires étudiés durant l'expérimentation. Celles-ci peuvent être l'efficacité, la fiabilité, etc.

Perspective La perspective indique à partir de quel point de vue les résultats de l'étude doivent être interprétés.

Contexte Le contexte est l'environnement dans lequel l'expérimentation a lieu. Celui-ci définit le type recherché de sujets pour conduire l'expérimentation ainsi que le type d'artéfacts avec lesquels les sujets devront interagir. Les sujets sont caractérisés par leur expérimentation, leur charge de travail, etc. Les artéfacts sont caractérisés par leur taille, leur complexité, leur domaine d'application, etc.

1. Le modèle GQM est un modèle de définition orienté but.

Le contexte peut être caractérisé par le nombre de sujets et le nombre d'artéfacts de l'expérimentation comme nous le montre la figure 5.2.

		# Objects	
		One	**More than one**
# Subjects per object	**One**	Single object study	Multi-object variation study
	More than one	Multi-test within object study	Blocked subject-object study

FIGURE 5.2 – Caractérisation du contexte

Tous ces types d'expérimentation peuvent être conduits comme des expérimentation ou des *quasi-expérimentation*. Une expérimentation se déroule quand les sujets et les objets de l'étude sont choisis de manière aléatoire tandis qu'une *quasi-expérimentation* s'effectue quand il y a un manque d'aléatoire dans le choix des sujets ou dans le choix des objets de l'étude [2].

5.2 Définition concrète de notre expérimentation

Nous pouvons, maintenant, passer à la définition concrète de notre expérimentation suite aux rappels théoriques que nous avons relevés à la section 5.1. Dans un premier temps, nous présenterons l'idée initiale qui nous a conduit à mener cette expérimentation et dans un deuxième temps, nous définirons l'expérimentation formellement.

5.2.1 Idée intuitive de l'expérimentation

Lors d'une tâche de maintenance, les développeurs déterminent les fichiers qui sont importants à comprendre et ceux qui sont importants à modifier pour effectuer une tâche. Pour cela, ils vont chercher dans les sources externes comme les listes d'emails, les *bugzilla*, les messages de commit[40] mais aussi dans les sources internes comme le code source et la documentation, en se posant des questions sur la structure et le comportement du système[2, 19].

2. La différence entre les expérimentation et les *quasi-expérimentation* est discutée en profondeur dans Robson[6]

De plus, les mainteneurs cherchent leurs informations dans un sous-ensemble de l'information totale du projet. Plusieurs outils ont été créés à partir de ce constat comme NavClus [3], Mylyn [4] et NavTrack [5].

Après avoir remarqué cela dans la littérature, nous avons voulu tester ces affirmations. En effet, nous voulions savoir comment les développeurs faisaient pour déterminer si un fichier [6] était important ou non pour une tâche de maintenance. Nous avons voulu aussi vérifier s'il y a une différence entre un fichier important à comprendre pour effectuer la tâche et un fichier important à modifier pour réaliser cette dernière. En outre, nous avons voulu vérifier si les mainteneurs cherchaient effectivement les fichiers pertinentes dans un sous-ensemble de toute l'information disponible pour résoudre une tâche.

5.2.2 Définition formelle de l'expérimentation

Nous nous sommes concentrés sur l'étude de 4 projets Java de tailles différentes dans un contexte de résolution d'une tâche de maintenance par des développeurs ayant une expérimentation variable allant du professionnel à l'étudiant. Sur ces projets, nous avons demandé aux développeurs d'effectuer une tâche de maintenance avec une brève description de celle-ci [7]. Nous avons essayé de voir comment ils pouvaient déceler les fichiers pertinents par rapport à leur tâche.

Ces éléments peuvent être résumés par le modèle suivant :

Analyser la résolution d'une tâche

dans le but de catégoriser/déceler les fichiers

en fonction de leur pertinence vis-à-vis de la tâche

du point de vue du développeur

dans le contexte d'étudiants et de professionnels sur 4 projets Java de tailles et complexités différentes.

3. https ://code.google.com/p/navclus/
4. http ://www.eclipse.org/mylyn/
5. L'outil est présenté dans [20]
6. Un fichier est tant du code source que de la documentation et des tests.
7. Les tâches furent définies à l'avance et leurs descriptions sont accessibles dans les annexes à la section C

Chapitre 6

Planification de l'expérimentation

Après la définition de notre expérimentation au chapitre 5 et plus particulièrement à la section 5.2.2, prend place la phase de planification. Cette dernière sert à dire comment va se dérouler notre expérimentation contrairement à la définition qui dit pourquoi une expérimentation est conduite.

En effet, comme dans toutes les activités liées à l'ingénierie, une expérimentation doit être planifiée et ses plans doivent être suivis pour pouvoir contrôler le processus. De plus, si une expérimentation n'est pas planifiée ni contrôlée correctement, les résultats peuvent être perturbés voire faussés, ce qui la rend inutilisable dans le pire des cas comme nous l'avons déjà évoqué à la section 5.1.

Pour expliquer cette planification, nous allons présenter brièvement la théorie reliée à cette phase dans la section 6.1. Cette théorie est issue de Wohlin[7]. Ensuite, nous mettrons cette théorie en pratique à la section 6.2.

6.1 Phase de la planification d'une expérimentation : rappels théoriques

La phase de planification d'une expérimentation peut être divisée en 7 étapes, comme nous le montre la figure 6.1, que nous expliquerons dans la suite de ce chapitre.

Ces étapes se déroulent selon un processus itératif jusqu'à ce qu'une planification complète soit prête. Cette dernière doit être préparée minutieusement car une fois qu'elle est faite, nous passons à la phase d'opération où il n'est plus possible de revenir en arrière. Comme nous l'avions expliqué au

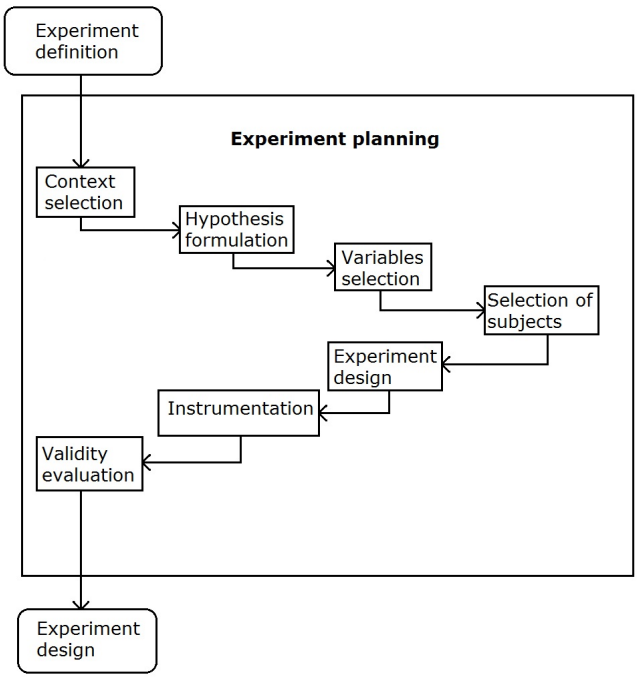


FIGURE 6.1 – Planification d'une expérimentation

chapitre 4, le processus complet se fait selon un modèle en spirale ce qui fait qu'à partir de cette étape, nous pouvons quand même revenir à la phase de définition des objectifs mais plus à partir de la phase d'opération.

Chaque étape est expliquée ci-dessous. En effet, la sélection du contexte est expliquée à la section 6.1.1, la formulation d'hypothèses à la section 6.1.2, la sélection des variables à la section 6.1.3, la sélection des sujets à la section 6.1.4, le design de l'expérimentation à la section 6.1.5 et l'instrumentation à la section 6.1.6.

6.1.1 Sélection du contexte

Pour qu'une expérimentation puisse avoir des résultats généraux, elle doit être conduite par des professionnels sur des projets en production. Toutefois, une expérimentation engendre des risques qui peuvent ralentir la vitesse de résolution des tâches par les professionnels. Les entreprises, voyant d'un mauvais œil un retard dans une tâche, préfèrent passer une expérimentation sur des projets hors-ligne en même temps que sur des projets réels. Cela réduit les risques mais induit des coûts supplémentaires.

Une alternative est de faire passer l'expérimentation par des étudiants. Ces expérimentations sont moins onéreuses et plus contrôlables mais les étudiants n'ont pas la même expérimentation que des professionnels et elle est moins variée. De plus, les expérimentations passées par des étudiants ne sont pas souvent effectuées sur des projets en production et de taille réelle mais sur des projets de taille réduite hors-ligne pour respecter des contraintes de tailles et de temps.

Cette alternative implique un compromis à faire entre des expérimentations qui peuvent sortir des résultats généraux et des expérimentations qui sortent des résultats spécifiques à un certain contexte. L'alternative permet de caractériser le contexte d'une expérimentation en 4 dimensions. Celles-ci sont :

- Projets Hors-ligne contre Projets en Production
- Étudiants contre Professionnels
- Projets Réduits contre Projets Réels
- Résultats Généraux contre Résultats Spécifiques

6.1.2 Formulation d'hypothèses

La formulation d'hypothèses est la base pour effectuer des analyses statistiques. Quand une hypothèse est formellement formulée et que les données sont collectées, l'hypothèse peut être acceptée ou rejetée. Si l'hypothèse peut être rejetée, nous pouvons en tirer des conclusions.

Durant la phase de planification, la définition de l'expérimentation est formalisée en 2 hypothèses que sont l'hypothèse nulle et l'hypothèse alternative.

Hypothèse nulle, H_0 Une hypothèse nulle dit qu'il n'y a pas de patterns dans les paramètres de l'expérimentation et que les différences observées sont dues à des coïncidences. Cette hypothèse est celle qui doit être rejetée avec le plus haut degré de vraisemblance.

Hypothèse alternative, $H_1, H_\alpha, H_\beta, etc.$ Cette hypothèse est celle qui est en faveur du rejet de H_0.

Avec ces 2 hypothèses, nous pouvons effectuer différents tests statistiques pour évaluer les données de l'expérimentation. Ils sont basés sur les hypothèses ci-dessus et doivent donc être choisis et réalisés après avoir formulé nos hypothèses. Le fait de tester nos hypothèses inclut 2 types de risques que sont le rejet d'une hypothèse vraie et l'acceptation d'une hypothèse fausse. Ces 2 risques sont connus sous le nom d'*erreur de type I* et d'*erreur de type*

II. Une *erreur de type I* apparait quand un test indique un pattern ou une relation alors qu'il n'y en a pas. Une *erreur de type II* apparait quand un test ne détecte pas de pattern ou de relation alors qu'il y en a 1. Le premier type d'erreur est exprimé de la manière suivante :

$$P(type - I - error) = p(rejectH_0|H_0true) \tag{6.1}$$

Le deuxième type d'erreur est caractérisé de la manière suivante :

$$P(type - II - error) = p(notrejectH_0|H_0false) \tag{6.2}$$

6.1.3 Sélection des variables

Avant de pouvoir lancer le design de notre expérimentation, nous devons choisir les variables indépendantes et dépendantes.

Choix des variables indépendantes

Les variables indépendantes sont celles que nous pouvons contrôler et modifier lors de l'expérimentation, comme nous l'avons expliqué au chapitre 4. Ces variables doivent avoir un effet sur les variables dépendantes. De plus, ces deux types de variables sont choisis simultanément ou en ordre inverse [1]. Le choix des variables indépendantes affecte aussi les échelles de mesure, la portée et les niveaux de celles-ci lors de l'expérimentation.

Choix des variables dépendantes

Les variables dépendantes permettent de calculer l'effet des traitements posés par l'expérimentation. La plupart du temps, une et une seule variable dépendante est choisie et provient directement de l'hypothèse. La variable n'est pas mesurable directement mais doit utiliser une mesure indirecte. Cette mesure affecte le résultat de l'expérimentation et doit donc être choisie précautionneusement. Le choix des variables dépendantes signifie que l'échelle de mesure et les portées des variables ont été déterminées.

6.1.4 Sélection des sujets

La sélection des sujets est une étape importante pour généraliser les résultats d'une population choisie. Les sujets sont choisis dans un échantillon probabiliste ou non. La différence entre ces 2 types d'échantillon se marque ainsi : dans le premier, chaque sujet est connu et dans le deuxième, chaque sujet est inconnu.

1. Nous choisissons d'abord les variables dépendantes puis nous prenons les variables indépendantes

Nous allons montrer 5 exemples de choix d'échantillonnage des sujets que sont le *Simple Random Sampling*, le *Systematic Sampling*, le *Stratified Random Sampling*, le *Convenience Sampling* et le *Quota Sampling*.

Simple Random Sampling Les sujets sont choisis aléatoirement à partir d'une population.

Systematic Sampling Le premier sujet est choisi aléatoirement à partir d'une population puis chaque $N^{ième}$ personne est choisie de cette population.

Stratified Random Sampling La population est divisée en plusieurs groupes dont la distribution est connue et un *Simple Random Sampling* est ensuite appliqué au sein de chaque groupe.

Convenience Sampling Les personnes les plus pratiques de la population sont choisies comme sujets.

Quota Sampling Les sujets sont choisis à partir de divers éléments de la population. Cet échantillonnage utilise le *Convenience Sampling* pour chaque sujet.

Comme nous le verrons à la section 6.2.4, nous avons choisi le *Simple Random Sampling*. Nous allons expliquer cette option plus en détail. Dans ce type de choix d'échantillonnage, chaque sujet a été choisi aléatoirement et entièrement par chance. Cette méthode est une technique de sondage impartiale et est un type d'échantillonnage de base car elle peut être utilisée comme un composant de méthodes plus complexes.

De plus, cet échantillonnage est souvent effectué *sans remplacement*, c'est-à-dire que nous évitons délibérément de choisir un sujet plus d'une fois pour une même expérimentation. En outre, elle est la plus simple des techniques d'échantillonnage et requiert le minimum de connaissances à l'avance sur la population étudiée. Dans le même temps, sa simplicité permet d'interpréter facilement les données collectées.

La taille de l'échantillon est aussi un facteur déterminant pour généraliser les résultats. En effet, plus l'échantillon est grand, plus le risque d'erreur diminue et inversement, plus l'échantillon est petit, plus le risque d'erreur augmente. Il est à noter que si l'échantillon reprend toute la population, il n'y a pas de risques d'erreur. Si la population a une forte variabilité, un échantillon plus large est nécessaire. De plus, la taille de l'échantillon est liée au test statistique qui sera utilisé. Le choix de la taille d'échantillon est influencé par l'analyse souhaitée des données. Il est donc important de savoir comment seront analysées les données à l'étape du design de l'expérimentation.

6.1.5 Design d'une expérimentation

Pour avoir des résultats significatifs à partir d'une expérimentation, nous utilisons des tests statistiques sur les données collectées pour interpréter les résultats mais les analyses statistiques qui peuvent être faites dépendent du design de l'expérimentation et des échelles de mesures utilisées.

Pour effectuer le design d'une expérimentation, nous devons regarder les hypothèses que nous avons formulées pour savoir quelles sortes d'analyses statistiques nous pouvons utiliser pour rejeter l'hypothèse nulle. Durant le design, nous déterminons combien de tests doivent être faits pour valider que l'effet du traitement soit visible. De plus, un bon design apporte les bases pour répliquer facilement l'expérimentation.

Cette section va se diviser 2 parties. Celles-ci sont les principes généraux d'un design et les types standards de design.

Principes généraux

3 principes doivent être pris en compte lors du design d'une expérimentation : l'aléatoire, le bloquant et l'ajustement.

L'aléatoire est l'aspect le plus important lors du design car toutes les méthodes statistiques l'utilisent pour s'assurer que les observations soient issues de variables indépendantes aléatoirement. L'aléatoire s'applique sur l'allocation des objets de l'étude, sur les sujets et sur l'ordre dans lequel les tests doivent être faits. Comme dit à la section 6.1.4, l'aléatoire sert aussi à choisir les sujets à partir d'une population.

Un effet bloquant survient en présence d'un facteur ayant un effet non recherché. Si le facteur est connu et contrôlable, nous pouvons l'éliminer systématiquement pour examiner, de manière fiable, les effets des traitements. Cette technique rend les comparaisons d'effets entre les différents traitements impossibles et ne sont donc pas étudiés. Elle permet aussi d'améliorer la précision de l'expérimentation.

L'ajustement est effectué pour que chaque traitement ait le même nombre de sujets. L'ajustement permet de simplifier l'analyse statistique des données mais n'est pas nécessaire pour faire une expérimentation.

Types standards de design

Nous allons maintenant présenter différents types de design qui sont appropriés pour des expérimentations ayant *un facteur avec 2 traitements*, *un*

facteur avec plus de 2 traitements, *2 facteurs avec 2 traitements* et *plus de 2 facteurs avec chacun 2 traitements*

Dans le type d'expérimentation où nous avons un facteur avec 2 traitements, l'intérêt est seulement de comparer deux traitements. Cette comparaison se fait sur la variable dépendante de chaque traitement. Il existe 2 designs différents pour ce type. Ils sont le *Completely randomized design* et le *Paired Comparaison design* et utilisent les notations de μ_i [2] et γ_{ij} [3].

Dans le type d'expérimentation avec un facteur ayant plus de 2 traitements, nous voulons comparer les traitements entre eux comme pour les expérimentations ayant un facteur pour 2 traitements. Il existe 2 types de designs différents : le *Completely randomized design* et le *Randomized complete block design*. Le premier type est déjà défini dans les expérimentations avec 2 traitements pour 1 facteur.

La complexité de l'expérimentation augmente quand nous passons d'un facteur à 2 facteurs ayant chacun 2 traitements parce que l'hypothèse doit être divisée en 3 hypothèses. En effet, nous avons une hypothèse pour l'effet sur un des facteurs, une deuxième hypothèse pour l'autre facteur et une troisième hypothèse pour tester l'interaction entre les 2 facteurs. Il existe 2 types de designs pour ce type d'expérimentation, le 2×2 *factorial design* et le *2-stage nested design*, avec les notations de τ_i pour l'effet du traitement i sur le facteur A, de β_j pour l'effet du traitement j sur le facteur B et $(\tau\beta)_{ij}$ pour l'effet de l'interaction entre τ_i et β_j.

Dans beaucoup de cas, l'expérimentation joue avec plus de 2 facteurs rendant alors l'effet de la variable dépendante non plus dépendante de chaque facteur séparément mais aussi de l'interaction entre les facteurs. Ces interactions peuvent se produire entre 2 traitements ou plus. Nous allons présenter 2 designs, le 2^k *factorial design* et le 2^k *fractionnal factorial design*.

6.1.6 Instrumentation d'une expérimentation

Les instruments sont choisis lors de la planification d'une expérimentation. Ils sont de 3 types : les objets, les guides et les mesures. Avant d'arriver à la phase d'opération, les instruments sont développés spécifiquement pour l'opération visée. Le but général de l'instrumentation est de fournir le moyen de réaliser la collecte des données de l'expérimentation. Cette récolte de données doit se réaliser sans affecter le contrôle de l'expérimentation. Les

2. La moyenne de la variable dépendante pour le traitement i.
3. La j^{eme} mesure de la variable dépendante pour le traitement i.

résultats doivent être indépendants de l'instrumentation. Si celle-ci affecte l'aboutissement de l'expérimentation, les résultats sont invalides.

Les objets de l'expérimentation peuvent être de la documentation, des spécifications, du code source, etc. Le choix des différents objets est important pour avoir des objets appropriés par rapport à l'expérimentation que nous voulons réaliser.

Les guides de l'expérimentation sont utilisés pour mener le sujet lors de l'expérimentation. Si différentes méthodes sont utilisées, des guides doivent être créés pour chacune. De plus, chaque sujet devra avoir un entrainement dans ces méthodes. Les guides sont, par exemple, des descriptions du processus de l'expérimentation et du déroulement de celle-ci.

Les mesures sont réalisées par la collecte des données. La planification sert à préparer les formulaires et les questionnaires d'interview qui seront donnés aux sujets. Ceux-ci seront distribués à certaines personnes ayant une connaissance des expérimentations pour les valider.

6.2 Planification concrète de notre expérimentation

Après avoir montré le cadre théorique de la planification d'une expérimentation, nous pouvons enfin présenter la nôtre. Nous suivrons la même découpe en sous-sections qu'à la section 6.1.

6.2.1 Sélection du contexte

Nous avons choisi de faire passer l'expérimentation par des étudiants sur des systèmes hors-ligne mais réels pour essayer de faire ressortir des résultats généraux. Le choix de prendre une majorité d'étudiants s'explique par le fait que des étudiants étaient disponibles et que ceux-ci étaient déjà dans les locaux où se passait l'expérimentation. Nos 4 projets Java sont de vrais projets de taille réelle dont nous avons pris une version antérieure pour connaître la solution aux tâches de maintenance demandées. Ces projets sont *ECF*[4], *PDE*[5], *jEdit*[6] et *JhotDraw*[7]. Les 2 premiers sont considérés comme

4. Eclipse Communication Framework. Voir `http://www.eclipse.org/ecf/` pour plus d'informations

5. Plug-in Development Environment. Voir `http://www.eclipse.org/pde/` pour plus d'informations.

6. Un éditeur de texte. Voir `http://www.jedit.org/pourplusd'informations.` pour plus d'informations.

7. Un framework pour les GUI. Voir `http://www.jhotdraw.org/` pour plus d'informations.

des projets multi-modules où chaque module est un projet Java tandis que les deux derniers sont de simples projets Java. Nous avons donné aux sujets des documents décrivant ces projets. Ils sont présentés, dans les annexes, à la section A.

Nous avons donc un contexte avec les dimensions suivantes :
- Projets Hors-ligne
- Majorité d'étudiants
- Projets Réels
- Résultats Généraux

De plus, les projets ont été choisis selon 4 critères : la disponibilité du code, le nombre de développeurs, le domaine d'application et la disponibilité des données. Concernant la disponibilité du code, il devait être open-source pour avoir un accès à un *repository* SVN ou GIT permettant de récupérer les projets avec leurs différentes versions et les messages des développeurs. Pour le nombre de développeurs, nous nous étions imposés comme seule contrainte qu'un projet devait être écrit par plusieurs personnes. Nous avons aussi sélectionné des projets ayant un domaine d'application différent des autres projets choisis. Pour la disponibilité des données, les projets retenus avaient déjà été utilisés dans des expérimentations précédentes. Ceci nous a permis d'obtenir des données concernant les différents bugs des projets.

Les tâches que nous avons déterminées reprennent 2 des différents types de maintenance expliqués à la section 3.3. En effet, nous effectuons des maintenances correctives pour les projets *ECF*, *PDE* et des maintenances adaptatives pour les projets *jEdit* et *JHotDraw*. Ces maintenances s'effectuent toutes selon le modèle *quick-fix*, expliqué à la table 3.6.1, car il est le plus adapté pour des maintenances correctives et adaptatives simples[22].

Nom de la tâche	Description de la tâche
ECF	La commande *Select All* ne fonctionne pas correctement sur le *channel* IRC
PDE	La valeur *Autostart* n'est pas stockée correctement dans le fichier *product*
jEdit	Ajouter les numéros de lignes et des messages d'erreur dans la boite de dialogue permettant la sélection d'une ligne
JHotDraw	Changer les couleurs des labels et du background de l'application *FontChooser*

TABLE 6.1 – Résumé des tâches de maintenance

6.2.2 Formulation d'hypothèses

Notre hypothèse nulle est qu'il est impossible de détecter les fichiers pertinents pour une tâche donnée. Si nous pouvons rejeter cette hypothèse, nous pourrons sortir des résultats. Si nous pouvons rejeter le fait qu'il est impossible de trouver à l'avance les fichiers pertinents pour une certaine tâche, alors nous pouvons déceler les fichiers pertinents à l'avance. C'est ce que nous allons tenter de prouver par cette expérimentation.

De plus, cette hypothèse prend place dans un contexte où nous voyons la maintenance d'un logiciel comme un processus incrémentiel où nous nous servons des maintenances précédentes pour nous aider à trouver les fichiers pertinents pour effectuer la maintenance actuelle. Dans cette vision de la maintenance, certains modèles, que nous avons vus à la section 3.6, sont plus appropriés que d'autres pour effectuer une maintenance en utilisant la connaissance incrémentale. Ceux-ci sont en particulier le modèle d'amélioration itérative [8] et le modèle de Boehm [9] parce qu'ils peuvent être modifiés facilement pour inclure les traces des maintenances passées en vue de déceler les fichiers pertinents.

En plus de cette hypothèse nulle, nous avons deux hypothèses alternatives :

1. Certains fichiers sont significativement pertinents [10].
2. Certains fichiers sont hautement pertinents [11].

6.2.3 Sélection des variables

Nos variables indépendantes concernent tout ce qui est lié à l'environnement expérimental. En effet, tout ce qui touche aux projets et aux tâches est contrôlé et fixé avant chaque passage de sujet. Ceux-ci sont aussi contrôlés pendant le passages des sujets.

Notre variable dépendante est l'ensemble des interactions commises par un sujet lors de la résolution de sa tâche.

6.2.4 Sélection des sujets

La participation volontaire des sujets est avérée : ils n'ont pas été forcés de participer à l'expérimentation. Nous avons envoyé un questionnaire de recrutement sans demander d'informations limitant l'accès d'une personne à

8. Voir section 3.6.2
9. Voir section 3.6.4
10. La notion de significativement pertinent est donnée au chapitre 1.
11. La notion de hautement pertinent est donnée au chapitre 1.

l'expérimentation. C'est pourquoi, nous pouvons dire que nous avons effectué un choix *Simple Random Sampling* où notre population est la population des développeurs et nous ne pouvions prévoir le nombre ou l'ordre d'apparition des candidats pour l'expérimentation. De plus, nous avons choisi la méthode *sans remplacement* car nous avons délibérément évité de faire qu'un sujet passe 2 fois notre expérimentation.

Notre questionnaire nous permettait de pouvoir aussi affecter une tâche à chaque sujet. L'ensemble des tâches est ainsi réparti dans une population la plus variée possible selon les critères de niveau d'étude, de sexe, ou de nombre d'années d'expérimentation en Java. La figure 6.2 montre la répartition des sujets selon les critères établis par le questionnaire.

Projet	Tâche	Identifiant du sujet	Sexe	Niveau d'études	Années d'expérimentation en Java
ECF	2	5	Femme	Doctorat	6
ECF	2	6	Homme	Doctorat	5
ECF	2	7	Femme	Doctorat	5
ECF	2	18	Homme	Doctorat	3
PDE	4	2	Homme	Master	3
PDE	4	13	Femme	Doctorat	6
PDE	4	14	Homme	Doctorat	9
PDE	4	15	Homme	Professionnel	15
PDE	4	17	Homme	Doctorat	1
jEdit	5	9	Femme	Doctorat	10
jEdit	5	12	Femme	Doctorat	5
jEdit	5	16	Homme	Doctorat	3
JHotDraw	6	10	Homme	Doctorat	1
JHotDraw	6	11	Homme	Doctorat	5
JHotDraw	6	8	Femme	Doctorat	5
JHotDraw	6	3	Femme	Doctorat	3

TABLE 6.2 – Répartition des sujets de l'expérimentation

6.2.5 Design de notre expérimentation

L'expérimentation fut, dès le départ, conçue avec un design misant, parmi les principes généraux, sur l'aléatoire et l'ajustement. En effet, nous avons partagé les sujets équitablement entre les différents traitements que nous avions à notre disposition. De plus, comme dit à la section 6.1.4, c'est l'aléatoire qui a conduit à la sélection des sujets mais c'est aussi cette propriété qui nous permet d'effectuer nos différentes analyses statistiques.

Pour le type standard de design, nous avons choisi le design comprenant 1 facteur avec plus de 2 traitements. En effet, notre facteur est la manière dont un développeur va explorer un système pour effectuer une tâche de maintenance. Le fait que chaque personne soit unique implique que chaque développeur ait une manière d'explorer qui lui soit unique. Ce fait implique qu'il y ait autant de traitements que de sujets.

6.2.6 Instrumentation de notre expérimentation

Dans notre cas, les sujets avaient accès à tout le code source des projets mais ne pouvaient utiliser ni les messages de commit ni le *bugzilla* d'Eclipse ni aller changer de version du projet. En effet, nous avons récupéré les projets et les avons mis sur le *SVN* de l'École Polytechnique de Montréal pour empêcher les sujets de trouver les solutions sans modifier le code et nous donner des résultats erronés. De plus, les projets choisis, déjà expliqués à la section 6.2.1, n'ont pas été modifiés par rapport aux versions disponibles sur les gestionnaires de versions. La seule différence avec les versions en production était que les sujets travaillaient sur des versions vieilles de 5 ans.

Concernant les guides, nous avons créé des documents présentant les projets, la tâche pour chaque projet et une procédure à suivre qui était similaire à tous les sujets pour ne pas induire de biais. La procédure est disponible à l'annexe E. En effet, en respectant la même procédure pour tous les sujets, l'expérimentateur ne peut donner plus ou moins d'informations aux sujets.

Pour les formulaires, nous avons préparé un formulaire d'inscription nous donnant des informations basiques sur les sujets comme leur adresse e-mail, leur disponibilité, leur niveau d'étude, leur expérimentation en Java et leur sexe. Nous avons aussi préparé un formulaire de pré-expérimentation qui reprend certaines questions du formulaire d'inscription comme le niveau d'études et l'expérimentation en Java mais ce questionnaire pose aussi des questions plus spécifiques quant à la connaissance de l'*IDE* Eclipse pour la programmation ainsi que la connaissance de la programmation de plugins et de framework pour PDE et ECF respectivement.

Chapitre 7

Opération de l'expérimentation

Dans ce chapitre, après certains rappels théoriques issus de Wohlin[7], nous présenterons la phase opérationnelle de l'expérimentation. En effet, une fois l'expérimentation bien définie, nous passons à une phase d'exécution de celle-ci avant de pouvoir analyser les résultats obtenus. Nous diviserons cette phase en 3 étapes : la préparation, l'exécution et la validation des données. La première étape permet de préparer un sujet afin qu'il exécute sa tâche sans y introduire de biais tel que l'absence de motivation. La seconde étape s'intéresse à l'exécution de la tâche. Enfin, la dernière étape consiste à vérifier que la tâche a été suffisamment bien effectuée et que les résultats peuvent être utilisés. Cette étape d'opération est représentée à la figure 7.1.

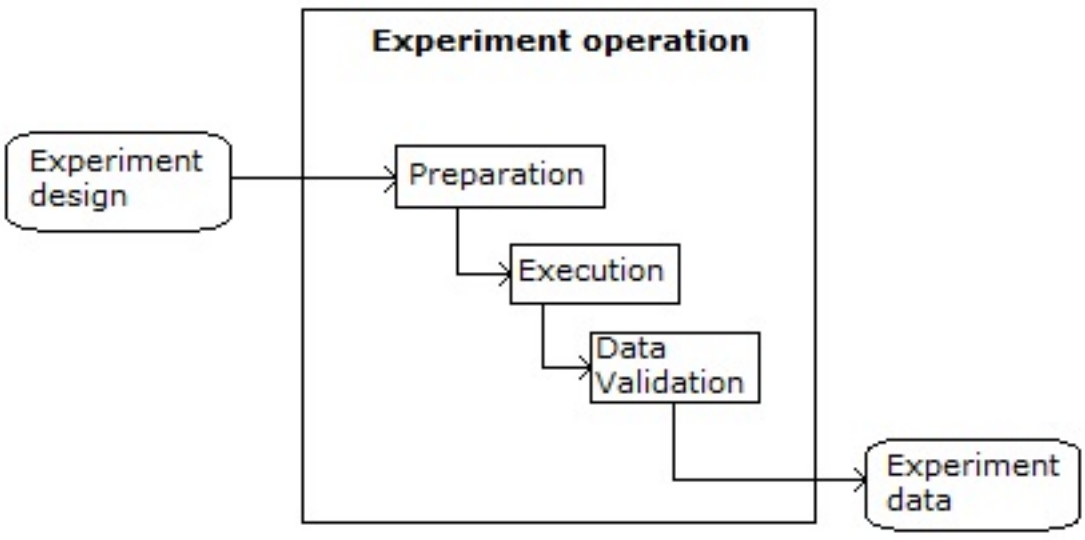


FIGURE 7.1 – Phase opérationnelle

7.1 Phase d'opération : rappels théoriques

7.1.1 Étape de préparation

Cette étape est la base de l'expérimentation. En effet, avant de commencer l'exécution d'une expérimentation, il faut préparer les participants ainsi que le matériel qui va être utilisé dans l'expérimentation. Nous verrons dans cette section ce qui a été mis en place lors de notre expérimentation.

Préparation des participants

Le choix des sujets a été expliqué dans la section 6.2.4. Il faut désormais s'intéresser aux différents risques liés au caractère des sujets sélectionnés.

Le consentement Il est important de citer nos objectifs et d'expliquer, sans révéler d'informations concernant l'expérimentation qui va être faite, les buts généraux de l'expérimentation pour en informer le sujet et lui montrer l'utilité de ce qu'il va accomplir. En effet, si le sujet n'est pas en accord avec les objectifs ou n'est pas décidé à accomplir sa tâche dans les règles et objectifs fixés par l'expérimentation, il y a un risque que ces données soient invalides.

L'anonymat Les expérimentations menées sur des développeurs peuvent collecter des données sensibles. Par exemple, collecter des informations concernant la productivité de programmeurs en milieu industriel revient à collecter des données très sensibles qui ne peuvent en aucun cas être associées à une personne.

La motivation La motivation est un élément important lors d'une expérimentation. C'est pourquoi un ensemble d'expérimentations sont rémunérées ou permettent d'obtenir un gain une fois la tâche réalisée. Cependant, ces gains peuvent aussi fausser l'expérimentation car les sujets risquent de ne le faire que pour ce gain.

La trahison Un dernier point est mis en avant par Wohlin[7] : la trahison. En effet, il est clair que plus le sujet est prévenu et conscient de l'expérimentation, plus l'expérimentation ne comporte pas de biais. Il est donc essentiel de prévenir le sujet, s'il est filmé, enregistré, et quelles données seront utilisées.

Préparation du matériel

La préparation du matériel est une chose très importante lors de l'exécution d'une expérimentation. En effet, le matériel ne doit en aucun cas perturber le sujet lors de son expérimentation pour que ses données puissent être validées. De plus, il faut que chacun des sujets soit traité de la même manière et reçoive pour chacune des tâches les mêmes documents qu'un autre participant. Si cela n'est pas respecté, l'expérimentation peut induire

des résultats dépendant des documents et non de l'expérimentation et du sujet lui-même.

7.1.2 Étape d'exécution

Cette phase est la phase la plus importante et elle comporte deux points importants qui risquent d'introduire des biais dans les résultats : l'environnement expérimental et la collecte de données.

L'environnement expérimental

L'environnement dans lequel se déroule l'expérimentation [1] se doit d'être le plus ressemblant par rapport à la réalité du projet. Il est donc important de ne pas modifier le projet sur lequel le sujet va devoir travailler. En effet, une modification risquerait de rendre le résultat dépendant d'un projet expérimental et donc de ne plus concerner le projet, qu'il soit industriel ou open-source, que l'on veut réellement étudier. Cependant certaines modifications peuvent être apportées si le projet est trop imposant ou contient des parties spécifiques qui pourraient induire en erreur le résultat et donc invalider les données.

La collecte de données

Plusieurs choix sont possibles pour collecter les données.

Penser à voix haute Cette méthode a d'abord été utilisée pour faire tester un système par un utilisateur. Ce principe a ensuite été utilisé pour de nombreuses expérimentations. Le principe consiste à demander aux sujets d'expliciter verbalement l'ensemble des actions qu'ils effectuent. L'examinateur peut alors prendre note des explications intéressantes pour l'expérimentation. Cependant, cette méthode a l'effet négatif de mettre en doute le sujet, et donc de perdre des interactions spontanées[18].

Formulaires Les formulaires à remplir sont largement utilisés. Le principe est de poser des questions par écrit ou oralement après que le sujet ait fini sa tâche.

Traçage Des outils sont disponibles pour tracer les actions des sujets lors d'une tâche de développement. La différence entre les outils se situe au niveau de la granularité de capture, certains vont uniquement traquer les changements dans les fichiers, d'autres dans les méthodes. Ensuite, ces outils permettent pour la plupart une visualisation particulière. Ayant nous-mêmes redéfini une interprétation des données à la section

1. Dans ce cas-ci, lorsque l'on parle d'environnement, on considère le projet sur lequel va se dérouler l'expérimentation

7.2.2, nous citerons simplement ici trois outils connus de traçage : NavTrack[20], NavClus[37] et Mylyn [2]

Capture vidéo Pour capturer les informations, il est possible d'enregistrer le sujet. Pour ne pas stresser les participants, il est possible via certains logiciels d'enregistrer uniquement l'écran.

Eye-tracking Cette méthode permet de capter les endroits où la vision du développeur s'arrête. Cependant, les logiciels sont coûteux et doivent être adaptés à chaque type d'événements que l'on désire capturer. De plus, cette technique demande un équipement qui peut troubler et intimider le sujet lors de son expérimentation.

7.1.3 Étape de validation des données

Les données enregistrées doivent être ensuite validées par l'examinateur. En effet, il se peut qu'un sujet ait mal compris la façon dont devait se dérouler l'expérimentation ou qu'il n'ait pas l'expérience nécessaire pour réaliser la tâche ou répondre aux questions. Ces cas sont souvent évités par le remplissage préliminaire d'un questionnaire qui permet aux examinateurs de savoir si le sujet sera capable d'effectuer l'expérimentation. Cependant, certains cas ne sont pas décelés et donc une analyse a posteriori doit être effectuée. La validation des données et de l'expérimentation est expliquée plus en détail au chapitre 7.2.3.

7.2 Opération concrète de notre expérimentation

7.2.1 Étape de préparation

Préparation des participants

Le consentement Les sujets de notre expérimentation recevaient la procédure à suivre et cette dernière contenait le but de l'expérimentation ainsi qu'une courte description du type de tâches qu'ils allaient devoir réaliser. Ceci permettait d'informer le sujet et de recevoir son accord sur la réalisation de la tâche [3].

L'anonymat Notre expérimentation ne collectait pas d'informations sensibles sur l'identité et la vie privée. De plus, les sujets étant pour la plupart issus du milieu universitaire, la résolution ou non de la tâche de maintenance ne comportait pas de frein à l'expérimentation. Cependant, nous avons tout de même choisi par respect pour nos sujets de garder l'anonymat de chacun d'eux. C'est pourquoi l'ensemble des fichiers de l'expérimentation contenait seulement les numéros attribués aux sujets. Les numéros affectés aux sujets se faisaient comme

2. `http://www.eclipse.org/mylyn/`
3. Le document *Experiment procedure* est visible en annexe D

suit : NomDuProjet_NuméroDeTâche_NuméroDeSujet. Le sujet numéro 19 de la tâche 2 sur le projet PDE aura donc l'appellation suivante pour conserver son anonyma t : PDE_4_s19 .

La motivation La motivation est la raison de notre choix d'utiliser, pour notre expérimentation, des sujets issus du milieu universitaire. Ceux-ci, par leur formation sont soucieux de la recherche et conscients des résultats que peuvent amener ce genre d'expérimentation.

La trahison Dans notre expérimentation, nous avons expliqué à nos sujets les deux captures que nous allions effectuer : la capture vidéo et la capture de trace *Mylyn*. Ceci nous permettait d'être totalement transparents. Ceux qui ne n'acceptaient pas les règles définies, en début d'expérimentation, pouvaient refuser d'effectuer celle-ci.

Préparation du matériel

Lors de notre expérimentation, pour éviter toute négligence, nous avons préparé une *check-list*[4] qui permettait de n'oublier aucun détail et de suivre une procédure identique pour chaque sujet. Chaque tâche était accompagnée d'un set de documents comme expliqué dans la section 6.2.6.

7.2.2 Étape d'exécution

L'environnement expérimental

Lors de notre expérimentation, nous avons gardé les projets PDE, ECF, jEdit et JHotDraw tels qu'ils étaient lorsque le bug a dû être résolu. L'environnement est donc recréé à l'identique par rapport à celui qui avait existé à l'époque où il fallut résoudre le bug sur le projet.

L'environnement de développement de l'expérimentation fut la distribution GNU /Linux CentOS 6.5 10 sur un machine avec un écran full HD. Sur cette machine étaient installées deux versions de l'IDE Eclipse : Eclipse 3.5.2 et 4.3. Ceci est dû au fait qu'une des tâches de maintenance de PDE ne fonctionnait pas sur Eclipse 4.3 qui était la dernière version disponible de l'*IDE* au moment de l'expérimentation, en raison de l'âge du bug relié aux tâches de maintenance. En effet, les bugs utilisés dans les tâches de l'expérimentation étaient anciens, de 2007 à nos jours. Le reste des tâches peut s'effectuer sur la version 4.3. Cependant, les différentes versions des projets Eclipse sont des versions spécifiques d'Eclipse.Les versions actuelles des projets concernés par les tâches étaient ensuite obtenues via leurs *repositories* SVN ou Git. Git fut utilisé pour les projets ECF et PDE tandis que SVN fut utilisé pour jEdit et JHotDraw.

4. Cette liste se trouve en annexe E

Une fois les versions des tâches récupérées, elles étaient stockées sur un *repository* SVN propre à l'expérimentation. Ceci nous assurait que l'environnement de l'expérimentation était identique pour l'ensemble des sujets. De plus, ceci nous assurait qu'aucun événement externe à l'expérimentation ne pouvait modifier les projets. Il n'était donc pas non plus possible pour les sujets d'obtenir des informations liées aux bugs sur les commentaires émis par les développeurs réels du projet dans les messages de commit sur les différents *repositories*. Une fois la tâche effectuée par un sujet, il suffisait donc de rétablir la version du projet ainsi que de vider l'ensemble des dossiers que l'*IDE* créait pour son fonctionnement.

L'ensemble des documents concernant les configurations requises pour l'expérimentation sont donnés en annexe F.1 et F.2.

La collecte de données

Le choix de la collecte des données est un choix important lors de la planification d'une expérimentation. Lors de notre expérimentation, plusieurs aspects ont dû être pris en compte : la facilité, la motivation des sujets, le temps de l'expérimentation.

Certaines techniques ne nous semblaient pas judicieuses. La durée moyenne de l'expérimentation prévue initialement était de 30 minutes, ce qui entrainait déjà un frein à la venue de nos sujets et une certaine sélection des tâches. Il ne nous était donc pas possible d'imposer la technique de penser à voix haute qui augmente durablement le temps de l'expérience ni de mettre en place un *eye-tracker* qui demande un certain temps de calibration et de nombreuses explications aux sujets.

Nous avons donc opté pour la capture d'écran, qui nous permettait de saisir l'ensemble des interactions et d'effectuer le travail d'analyse a posteriori sans gêner le sujet. Nous avons utilisé le logiciel VLC [5]. Celui-ci nous a permis de capturer l'ensemble des actions visibles à l'écran. La version de CentOS utilisée dans l'environnement expérimental ne nous permettait pas de capturer le pointeur de souris avec VLC car la version Linux du logiciel ne le permettait pas contrairement aux versions Windows et Mac OS. Mais ce logiciel est un bon choix de capture car il permet un encodage simple des vidéos et donc un gain d'espace disque sur les machines. Cette méthode a pourtant un biais qui est expliqué dans les menaces de l'expérimentation à la section 8.5.6.

5. Le logiciel est disponible à l'adresse suivante : `http://www.videolan.org/vlc/`

L'interprétation des données vidéos

Cette section explicitera la méthode que nous avons choisie pour interpréter nos données relevées par la capture vidéo. Nous avons défini huit éléments qui caractérisent une interaction et qui permettront ultérieurement d'analyser le comportement du sujet lors de la réalisation de sa tâche. Ces huit éléments sont :

Date de départ Représente la date de début de l'interaction. Celle-ci est exprimée sous la forme *hh :mm :ss*.

Date de fin Représente la date de fin de l'interaction. Celle-ci est exprimée sous la forme *hh :mm :ss*.

Nom de l'entité Représente le nom de l'entité de l'objet sur lequel l'interaction prend part. Si le chemin complet est disponible alors il est indiqué. La structure du chemin est la suivante : *Chemin/NomDuFichier. Extension*

Type d'entité Représente le type de l'objet. Ici, nous nous limitons à 3 types de granularité : projet, package, fichier. Le projet indique le nom du projet Java lorsque la tâche concerne plusieurs projets. Par exemple pour le projet global ECF et PDE, on distingue plusieurs sous-projets. Le package indique le dossier ou le package où le fichier est placé à l'intérieur du projet ou sous-projet. Le fichier indique le nom du fichier ainsi que son extension.

Origine Représente la partie de l'IDE Eclipse utilisée pour déclencher l'interaction. On distingue ici plusieurs origines possibles : L'Editor le Package Explorer, la Call Hierarchy View, le Search Result View, le Debug View et le Search Tool . Le package explorer permet l'exploration de l'ensemble des fichiers, la call hierachy view permet de connaitre l'ensemble des endroits dans le programme où une méthode est utilisée et de les exposer dans une vue spécifique. Il ne permet pas de voir l'implémentation de la méthode mais uniquement sa déclaration. Le search result est une vue qui montre, lorsque l'utilisateur effectue une recherche, l'ensemble des résultats obtenus par cette recherche. Le debug view est la vue qui permet lorsque l'utilisateur utilise le mode debug d'Eclipse ,de suivre des points d'arrêt dans l'exécution du logiciel et, ensuite, de naviguer entre chaque étape d'une séquence d'exécution pour connaitre l'exécution exacte du logiciel. La dernière vue est le Search tool qui est l'outil qui permet d'entrer une rechercher de texte dans l'ensemble du ou des projets.

Technique Représente la technique utilisée pour quitter l'interaction précédente. Elle permet donc de connaitre le moyen utilisé pour passer d'une interaction à une autre. On distingue plusieurs techniques : Visuelle, Réference statique, Recherche de texte, Debug. La technique visuelle représente le fait de cliquer sur le fichier et donc de n'utiliser

aucun élément automatique de l'IDE pour provoquer l'interaction. La technique de référence statique est la technique qui permet via l'outil automatique de l'IDE de trouver toutes les références à une méthode ou variable sélectionnée. La technique de recherche de texte est l'utilisation de l'outil de recherche de texte dans les fichiers. La technique de référence est l'utilisation de l'outil automatique qui permet de trouver toutes les références d'une méthode ou variable sélectionnée. Le débug est la technique qui consiste à instaurer des points d'arrêt dans le code source et d'ensuite naviguer selon l'exécution du programme testé entre ces points d'arrêt.

Action Représente le type d'interaction effectuée. On distingue ici plusieurs actions : Ouverture, Recherche, Run , Edit, Close et Selection. L'ouverture représente le fait d'ouvrir un fichier ou un package. La recherche représente le fait de lancer la recherche. L'exécution est l'action de tester le logiciel et donc de quitter l'IDE pour ré-effectuer la reproduction du bug. L'édition représente le fait de modifier en ajoutant, supprimant ou modifiant le code source d'un fichier. La fermeture représente la fermeture d'un fichier ou d'un package dans une des vues. La sélection correspond à la sélection de texte.

Commentaires Représente une information textuelle supplémentaire telle que le mot saisi dans une recherche. Ce champ n'est utile que si un problème de compréhension de l'interaction intervient.

L'ensemble des possibilités d'interprétation des données vidéos sont reprises dans le tableau 7.1.

L'ensemble de ces interactions sont collectées manuellement en regardant les captures vidéos d'un sujet. Celles-ci sont ensuite stockées sous forme de tableau et formatée avec la technique CSV [6] pour permettre l'analyse automatique de ces données.

6. CSV veut dire Comma Separated Values

Type	**Valeurs**
Date de départ	*hh :mm :ss*
Date de fin	*hh :mm :ss*
Nom de l'entité	*Chemin/NomDuFichier.Extension*
Type d'entité	projet paquet fichier
Origine	Editeur Explorateur de paquets Vue d'appels hiérarchiques Vue des résultats de recherche Vue de débugage Vue de l'outil de recherche
Technique	Visuelle Référence statique Recherche de texte Débugage
Action	Ouverture Recherche Exécution Édition Fermeture Sélection

TABLE 7.1 – Résumé des possibilités d'interprétation des données vidéos

7.2.3 Étape de validation des données

Lors de notre expérimentation, nous avons décidé d'établir plusieurs états de validation des données.

Réussite de la tâche Le sujet a réussi à résoudre le bug établi par sa tâche : les données sont donc validées.

Non-réussite de la tâche Le sujet n'a pas réussi à résoudre la tâche mais a compris les classes à modifier ou celles à comprendre pour effectuer la modification. Le fait de non-modifier n'est donc pas un problème de compréhension de la tâche et de l'environnement expérimental mais uniquement un problème de connaissance du langage Java : les données peuvent donc être validées

Non-réussite de l'expérimentation Le sujet n'a pas l'expérimentation requise pour effectuer la tâche. Soit il ne connait pas l'IDE Eclipse, ce qui empêche toute manipulation du code source, soit il ne connait pas le langage Java ou ne le connait que dans ses aspects théoriques : les données sont inutilisables.

L'ensemble des données récoltées ont permis de valider 15 sujets sur 19 répartis comme suit : 4 pour la tâche concernant le projet PDE, 4 pour la tâche concernant le projet JhotDraw, 4 pour la tâche concernant le projet ECF, 3 pour la tâche concernant le projet jEdit.

Chapitre 8

Analyses et interprétations

Après la phase opérationnelle de l'expérimentation, nous pouvons commencer l'analyse des données collectées lors de cette phase.

Nous diviserons ce chapitre en 3 parties. La première partie portera sur les rappels théoriques de la validité d'une expérimentation.

La deuxième partie de ce chapitre confirmera, par l'analyse des données globales des différents sujets, que lors d'une tâche de maintenance, le temps de recherche et de compréhension est bien plus grand que le temps de modification réel.

La troisième partie expliquera les différentes menaces qui pourraient intervenir dans la non-validité de nos résultats.

8.1 Validité de l'expérimentation : rappels théoriques

Pour que l'analyse des données puisse amener des résultats valides, nous devons discuter de la validité et des menaces de notre expérimentation qui peuvent amener des biais et fausser les résultats. Un rappel sur la validité est présenté dans cette section tandis que les menaces sont présentées à la section 8.5.

Une des questions fondamentales d'une expérimentation est de savoir si nos résultats sont valides. Et il est important de considérer cet aspect pour avoir une validité adéquate des résultats. Cette validité adéquate dit que les résultats doivent être valides pour la population considérée par l'expérimentation. En effet, une validité adéquate n'implique pas nécessairement une validité plus générale car une expérimentation conduite à l'intérieur d'une

organisation pour le compte de celle-ci peut donner des résultats valides seulement pour celle-ci et cela est suffisant. Mais d'un autre côté, si des conclusions plus générales doivent être tirées, la validité doit couvrir une portée plus générale également.

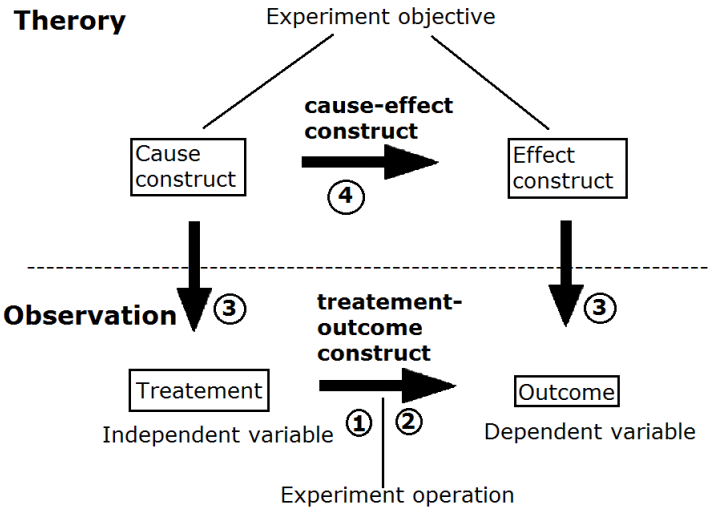


FIGURE 8.1 – Principes d'une expérimentation

Sur la figure 8.1, nous pouvons distinguer la partie reprenant la théorie et la partie reprenant l'observation. Nous voulons tirer des conclusions à propos d'une théorie. Cette théorie définit les hypothèses de l'expérimentation. En tirant des conclusions, nous franchissons 4 étapes. Ces étapes ont 1 type de menace quant à la validité des résultats.

La validité d'une expérimentation passe aussi par l'élimination des différentes menaces. Nous répertorions 4 types de validité : la validité interne, la validité externe, la validité de construction et la validité de conclusion. Ces types de validité se rapportent à différentes étapes d'une expérimentation et sont caractérisées par différentes menaces.

Validité de la conclusion Cette validité est concernée par la relation statistique entre le traitement et la conséquence du traitement. Cook[7] définit 7 types de menaces pour la validité de la conclusion. Ces 7 types sont : la puissance statistique basse, la violation d'hypothèses des tests statistiques, le taux d'erreur et de recherche, la fiabilité des mesures, la fiabilité de la

mise en place des traitements, les impertinences aléatoires et l'hétérogénéité aléatoire des sujets.

Validité interne Les menaces concernant la validité sont celles qui peuvent influencer la variable indépendante sans que l'expérimentateur ne le sache. Elles menacent la conclusion en donnant une possible relation entre le traitement et la conséquence. Elles se catégorisent en 3 groupes : les *menaces de groupes simples*, les *menaces de groupes multiples* et les *menaces sociales*.

Les menaces de groupes simples provoquent des problèmes pour déterminer si le traitement ou un autre facteur a causé l'effet observé. Cela arrive quand nous n'avons pas de groupe de contrôle auquel aucun traitement n'est appliqué. La plupart de ces menaces peuvent être gérées par le design de l'expérimentation. Ces menaces de groupe simple sont de 8 types : l'historique, la maturation, le test, l'instrumentation, la régression statistique, la sélection, la mortalité et l'ambiguïté.

Les menaces de groupes multiples existent parce que le contrôle du groupe et que la sélection des groupes peuvent varier d'un groupe à l'autre. Ces menaces sont identiques à celles décrites dans les menaces de groupes simples. En effet, lors de la sélection des groupes, ceux-ci vont être différents et ils auront, par exemple, une vitesse différente de maturation encore les groupes seront affectés différemment par l'historique, etc.

Les menaces sociales à la validité interne sont applicables autant aux groupes simples que multiples. Nous répertorions 4 types de menaces sociales : la diffusion ou imitation de traitements, l'égalisation compensatoire des traitements, la rivalité compensatoire et la démoralisation irritée.

Validité de la construction La validité de la construction est concernée par la généralisation des résultats à un concept ou une théorie. Nous relevons 2 types de menaces pour cette validité : les menaces de design et les menaces sociales. Chacun de ces types a plusieurs sous-types.

Les menaces de design contiennent 7 sous-types de menaces qui couvrent les problèmes de design et l'habileté de ce dernier à montrer clairement ce qui est étudié lors de l'expérimentation. Ces 7 sous-types de menaces sont les explications pré-opérationnelles inadéquates, le biais mono-opérationnel, le biais mono-méthode, la confusion entre les constructions et les niveaux de construction, l'interaction de différents traitements, l'interaction entre le test et le traitement et la généralisation restreinte des constructions.

Les menaces sociales sont liées au comportement des sujets et de l'expérimentateur. Elles sont basées sur le fait que ceux-ci peuvent agir différemment de leur mode opératoire habituel et donnent donc de faux résultats à l'expérimentation. Les menaces sociales ont 3 sous-types de menaces. Ceux-ci sont la devinette d'hypothèse, l'appréhension de l'évaluation et l'espérance de l'expérimentateur.

Validité externe La validité externe est menacée par des conditions qui limitent notre capacité à généraliser les résultats de notre expérimentation à une application commerciale. Il y a 3 types d'interactions avec des traitements qui peuvent amenuiser ce type de validité : l'interaction entre les échantillons et le traitement, l'interaction entre les paramètres et le traitement et l'interaction entre l'historique et le traitement. Les menaces concernant la validité externe sont réduites en utilisant un environnement aussi réaliste que possible.

8.2 Confirmation de l'importance du temps de recherche

Pour pouvoir confirmer que le temps de recherche et de compréhension est bien plus grand que le temps d'édition et de test, nous avons rassemblé les temps de l'ensemble des interactions qui correspondaient à une action recherche, édition ou exécution [1]. Nous avons rassemblé ces temps pour l'ensemble des sujets. Ensuite, nous en avons calculé la moyenne, en pourcentage, pour les comparer entre les différents projets. Ces temps sont rassemblés dans la figure 8.2.

La figure 8.2 montre que pour l'ensemble des tâches sur les projets ECF, jEdit,JHotDraw et PDE, le temps passé à chercher dans les explorateurs de fichiers ou dans l'éditeur sur un fichier lui-même a été pour l'ensemble des sujets l'activité principale. Cette activité est de l'ordre de 64% à 75% du temps global. On peut voir que la deuxième activité est le test pour tous les projets sauf JHotDraw mais aucun de ces temps ne dépasse les 30%.

On peut donc confirmer que le temps de recherche est significativement supérieur au temps d'édition. Il est donc intéressant de chercher à savoir comment réduire ce temps. Il est toutefois difficile, via l'analyse de vidéos, de distinguer le temps de recherche du temps de compréhension comme il est présenté à la section 8.5. Mais nous pouvons dire qu'une recherche fait partie de la compréhension vu qu'elle permet de connaitre la structure des modules et du projet en lui-même. De plus, comme le montre le graphique

1. Voir section 7.2.2 pour une explication de ces termes

8.3, nous avons pu déceler que le temps de recherche est réparti en grande majorité dans l'éditeur. On peut supposer que ce temps de recherche est un temps de compréhension du fichier.

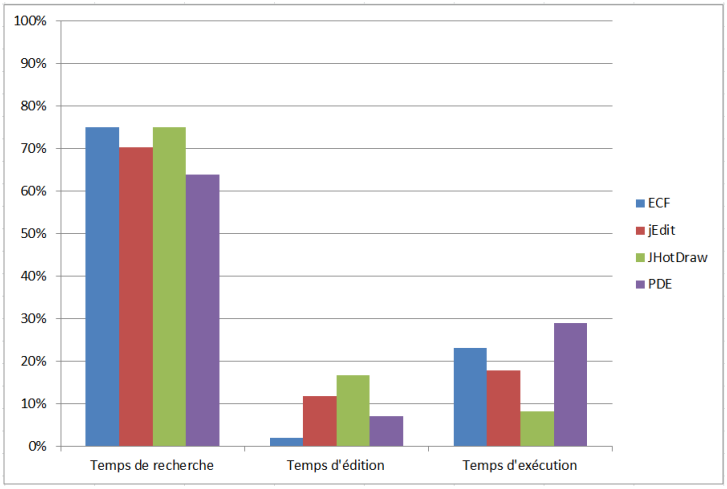


FIGURE 8.2 – Pourcentage des temps par projet. L'axe des abscisses représentent les temps de recherche, les temps d'édition, les temps d'exécution triés par projets. L'axe des ordonnées correspond au pourcentage de ces temps.

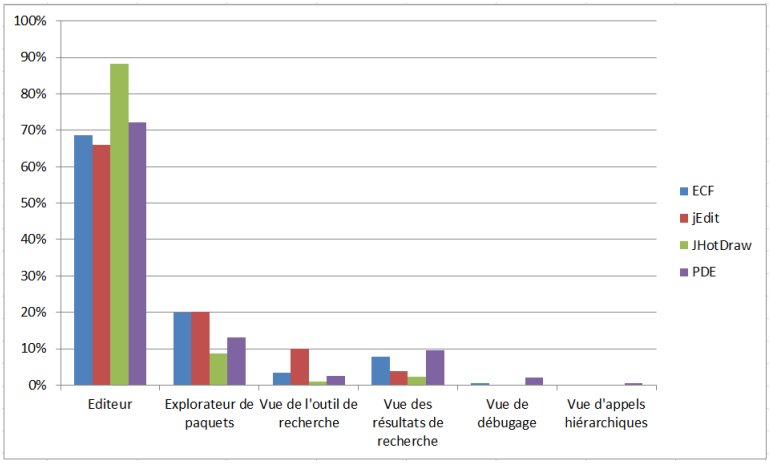


FIGURE 8.3 – Origine des recherches. L'axe des abscisses représente les différentes origines à partir desquelles une action recherche a été effectuée. L'axe des ordonnées correspond au pourcentage de ces origines.

8.3 Établissement des fichiers de références

Comme dit au chapitre 6, nos 4 tâches ont été reprises à partir d'un *repository* de bugs. Ces tâches sont donc réelles et ont été résolues par des développeurs du projets. Nous pouvons utiliser les informations de ces tâches comme fichiers de références. Les 2 sections 8.3.1 et 8.3.2 font la différence entre les fichiers significativement pertinents et les fichiers hautement pertinents.

8.3.1 Fichiers significativement pertinents

Pour rappel, un fichier significativement pertinent, pour Lee et Kang[37], est un fichier qui doit être changé pour accomplir la tâche de maintenance.

Grâce au *repository* de bug et aux informations comprises dans celui-ci, nous connaissons exactement le fichier à modifier ainsi que les modifications à l'intérieur de ce dernier. Le tableau 8.1 établit pour chaque tâche le nom du fichier à modifier pour résoudre celle-ci.

Projet	Identifiant de la tâche	Nom du fichier à modifier
ECF	2	ChatRoomManagerView.java
PDE	4	PluginConfigurationSection.java
jEdit	5	SelectLineRange.java
JHotDraw	6	PaletteFontChooserSelectionPanel.java

Table 8.1 – Ensemble des fichiers à modifier par projet

8.3.2 Fichiers hautement pertinents

Pour rappel, un fichier significativement pertinent, pour Lee et Kang[37], est un fichier qui permet de comprendre pourquoi et comment un fichier significativement pertinent doit être changé.

Les fichiers hautement pertinents pour les tâches de maintenance n'étaient pas disponibles et il est donc impossible d'obtenir ces fichiers de référence a priori. Sur base des données récoltées grâce aux sujets, nous essaierons de voir s'il n'est pas possible de définir ces fichiers.

8.4 Définition des critères de pertinence d'un fichier

Dans cette section, nous avons analysé les différents fichiers à notre disposition pour essayer d'en sortir des critères permettant de déceler les fichiers

hautement et significativement pertinents des autres fichiers d'un logiciel lors d'une phase de maintenance. Plusieurs axes d'analyses étaient disponibles, nous en avons sélectionné deux : l'axe du temps et l'axe des actions effectuées sur le fichier.

Nous avons obtenu nos chiffres en *parsant* les données des fichiers *CSV* qui ont les informations sur le temps passé dans les fichiers, le nombre d'actions par fichier, le nombre de techniques par fichier ainsi que l'origine de l'ouverture de ces fichiers et les techniques utilisées pour les ouvrir. Á partir de ces fichiers, nous avons ressorti de nouveaux fichiers *CSV* où nous ne gardons plus que les informations que nous jugeons pertinentes pour notre analyse. Ces informations sont expliquées dans leur section appropriée. En effet, nous avons sorti des informations pour le temps comme critère de pertinence à la section 8.4.1 et le nombre de revisites comme critère de pertinence à la section 8.4.2.

8.4.1 Le temps comme critère de pertinence

A partir des fichiers *CSV* où nous possédons les informations pertinentes pour le temps, nous avons extrait des graphiques sur 3 mesures différentes de temps : le temps total moyen passé par fichier, le temps de recherche moyen passé par fichier et le temps de modification moyen par fichier.

Le temps total passé par fichier

Pour calculer ce temps, nous avons additionné les temps de recherche et de modifications. Pour chaque sujet, nous mesurons le temps total passé dans chaque fichier. Pour pouvoir comparer les fichiers entre sujets affectés à une même tâche, nous avons transformé cette somme en pourcentage global par fichier.

Les graphiques 8.4, 8.5, 8.6 et 8.7 représentent la proportion du temps passé dans les fichiers d'un projet par l'ensemble des sujets. Ils représentent donc la moyenne de répartition du temps total par fichier.

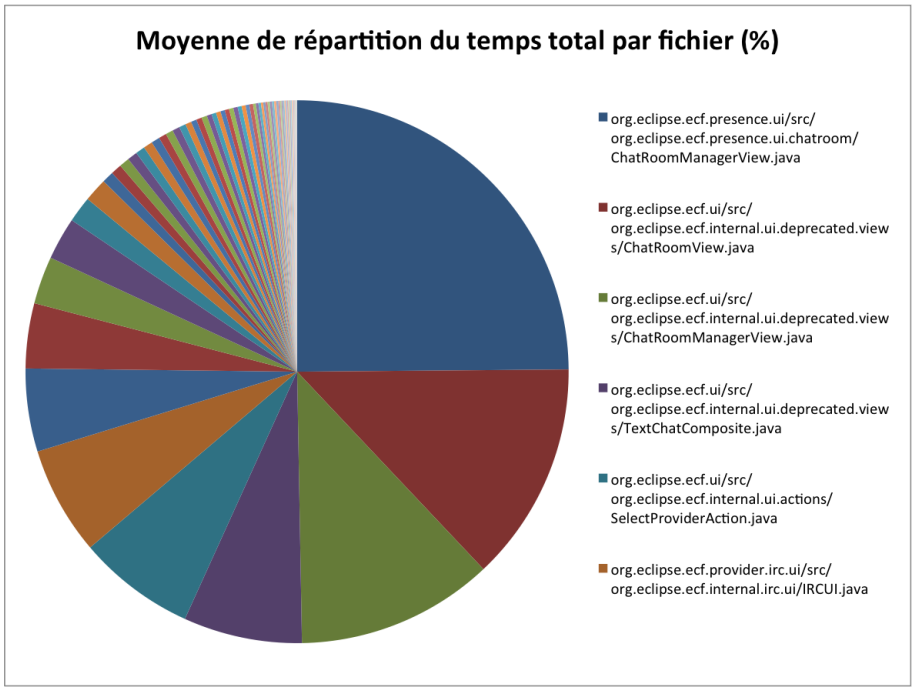


FIGURE 8.4 – Moyenne de répartition du temps total par fichier (%) pour ECF

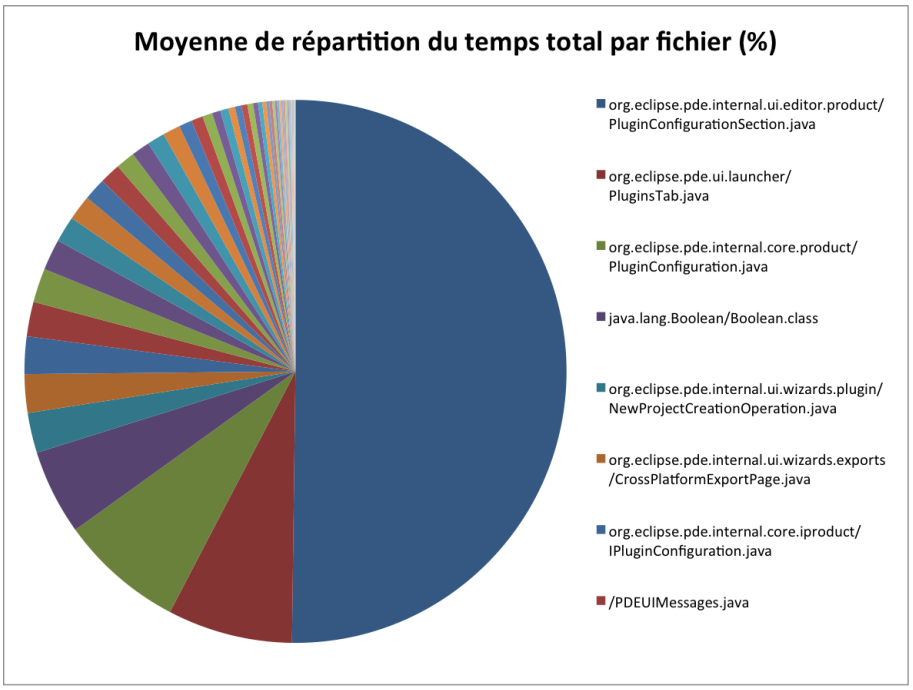


FIGURE 8.5 – Moyenne de répartition du temps total par fichier (%) pour PDE

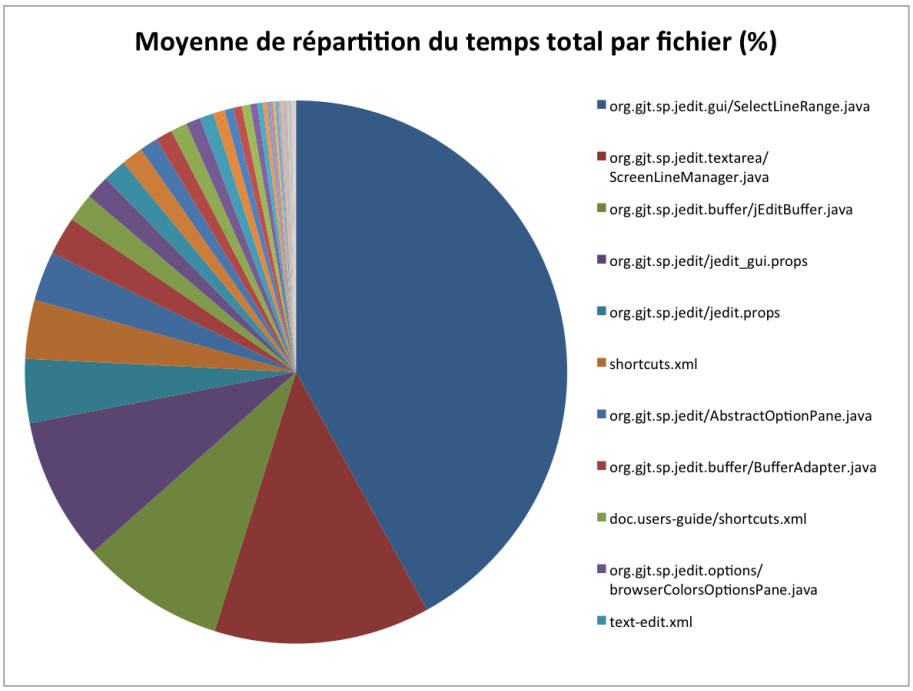


FIGURE 8.6 – Moyenne de répartition du temps total par fichier (%) pour jEdit

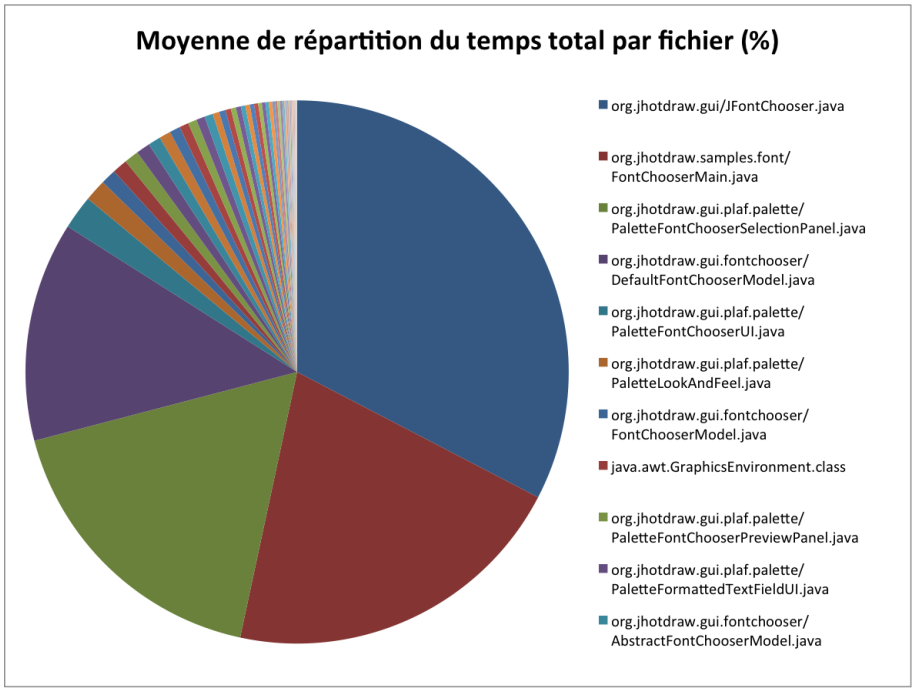


FIGURE 8.7 – Moyenne de répartition du temps total par fichier (%) pour JHotDraw

Les graphiques ont été réalisés grâce à la formule 8.1 qui a été appliquée à chaque fichier.

$$Temps_{cumulé d'un fichier} = \left(\frac{\sum_{i=1}^{N} \left(\frac{Temps_{fichier}(i)}{Temps_{ensemble fichiers}} \right)}{N} \right) \times 100 \qquad (8.1)$$

où N est le nombre de sujets sur un projet.

Des résultats de ces graphiques, nous pouvons déjà tirer quelques conclusions. Pour les 51 fichiers de PDE, les 4 fichiers où les testeurs ont passé le plus de temps représentent 69% du temps total, pour les 39 fichiers de jEdit, les 4 fichiers où les testeurs ont passé le plus de temps se partagent 73% du temps, pour les 51 fichiers de JhotDraw, les 4 fichiers où les testeurs ont passé le plus de temps représentent 81% du temps et pour les 79 fichiers d'ECF, les 7 fichiers où les testeurs ont passé le plus de temps se partagent 75% du temps [2]. Ces constatations nous conduisent à dire que le temps passé dans un fichier est une mesure appropriée pour déceler les fichiers pertinents. De plus, on peut voir que dans chaque projet, les fichiers permettant de résoudre la tâche, c'est à dire les fichiers significativement importants se retrouvent à chaque fois dans les fichiers dont le temps cumulé est le plus grand. On peut donc dire que un fichier significativement important est un fichier dont le temps cumulé est important.

Temps de recherche passé par fichier

Après avoir analysé le total global, nous allons essayer de voir si nous pouvons obtenir une meilleure précision ou la même précision avec les temps de recherche.

Les graphiques 8.8, 8.9, 8.10 et 8.11 représentent la proportion du temps de recherche passé dans les fichiers d'un projet par l'ensemble des sujets. Ils représentent donc la moyenne de répartition du temps de recherche par fichier.

2. Pour établir ces fichiers principaux, nous nous limitons au fichier dont le temps total par fichier est supérieur à 5% du temps total passé par tâche

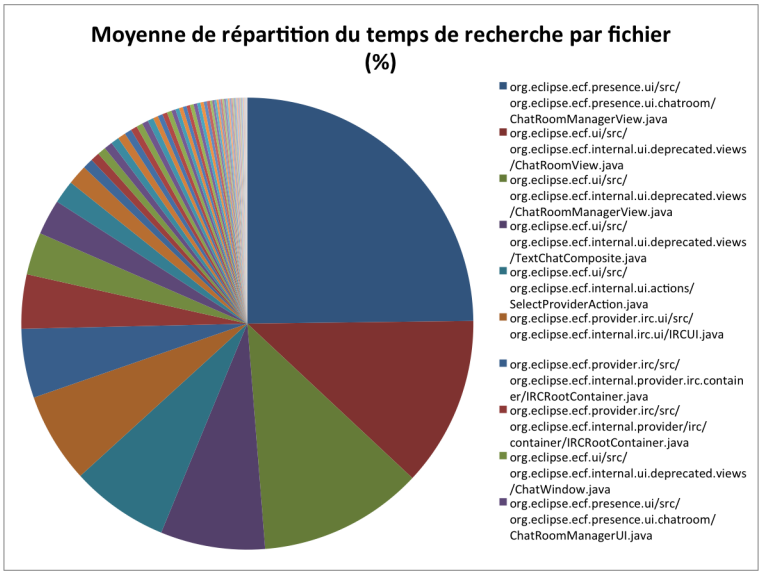


FIGURE 8.8 – Moyenne de répartition du temps de recherche par fichier (%) pour ECF

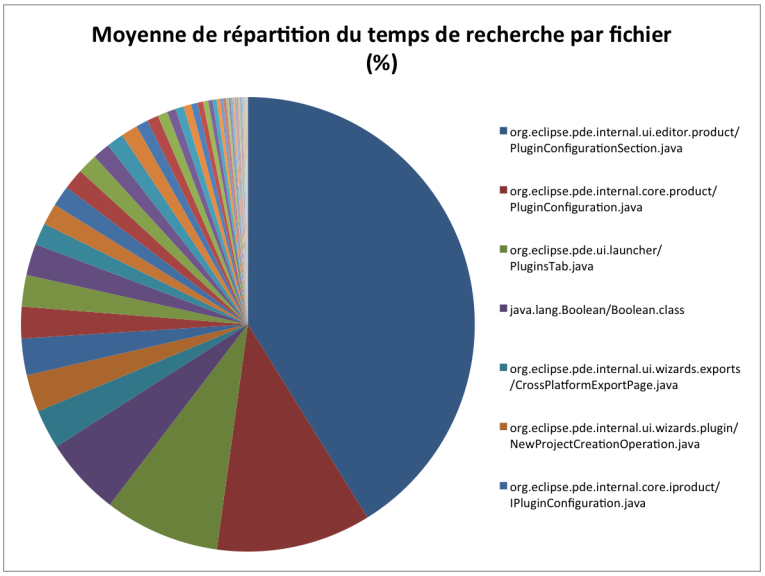


FIGURE 8.9 – Moyenne de répartition du temps de recherche par fichier (%) pour PDE

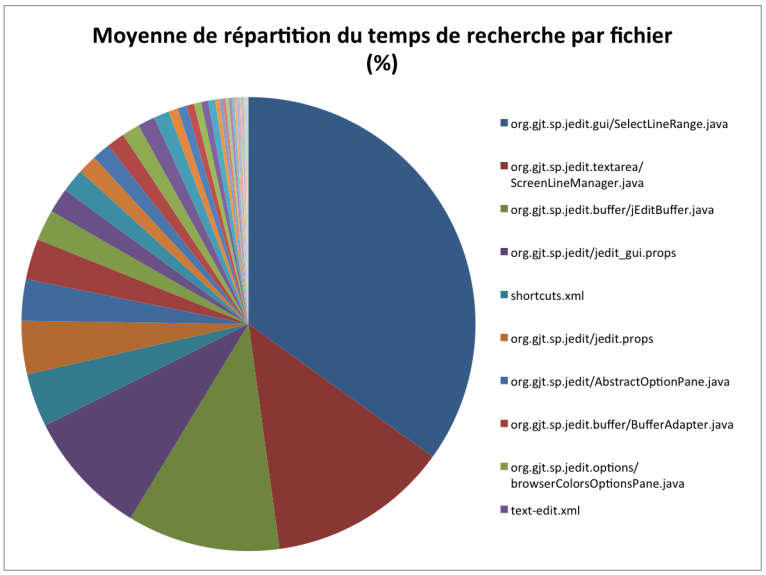


FIGURE 8.10 – Moyenne de répartition du temps de recherche par fichier (%) pour jEdit

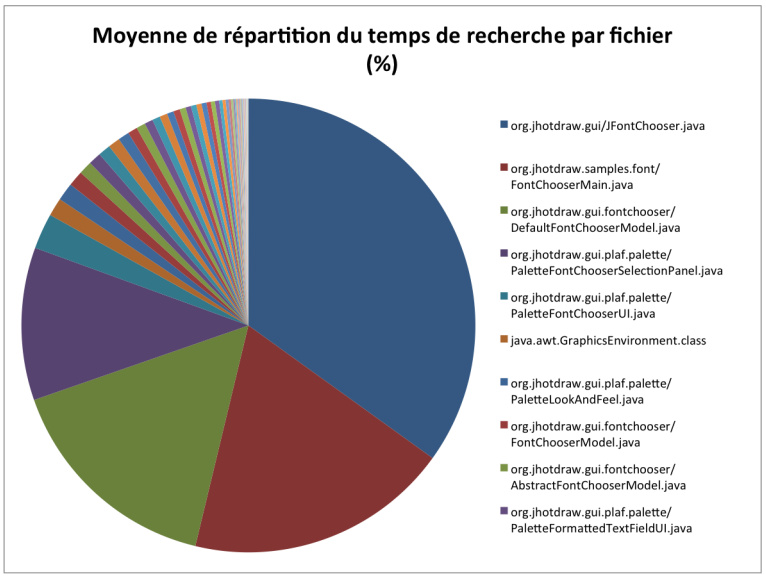


FIGURE 8.11 – Moyenne de répartition du temps de recherche par fichier (%) pour JHotDraw

Les graphiques ont été réalisés grâce à la formule 8.2 qui a été appliquée à chaque fichier.

$$Temps_{cumulé de recherche d'un fichier} = \left(\frac{\sum_{i=1}^{N} \left(\frac{Temps_{recherche}(i)}{Temps_{recherche ensemble fichiers}} \right)}{N} \right) \times 100 \quad (8.2)$$

où N est le nombre de sujet sur un projet.

Des résultats de ces graphiques, nous pouvons à nouveau tirer quelques conclusions. Pour les 51 fichiers de PDE, les 4 fichiers où les testeurs ont passé le plus de temps représentent 66% du temps total de recherche, pour les 39 fichiers de jEdit, les 4 fichiers où les testeurs ont passé le plus de temps se partagent 68% du temps, pour les 51 fichiers de JhotDraw, les 4 fichiers où les testeurs ont passé le plus de temps représentent 81% du temps et pour les 79 fichiers d'ECF, les 7 fichiers où les testeurs ont passé le plus de temps se partagent 74% du temps. Ces constatations nous conduisent à affirmer que le temps de recherche effectué dans un fichier est est une mesure appropriée pour déceler les fichiers pertinents [3]. De plus, on peut voir que dans chaque projet, les fichiers permettant de résoudre la tâche, c'est à dire les fichiers significativement importants se retrouvent à chaque fois dans les fichiers dont le temps de recherche cumulé est le plus grand. On peut donc dire qu'un fichier significativement important est un fichier dont le temps de recherche cumulé est important.

En outre, nous pouvons constater sur base de ces résultats que la précision diminue ou reste identique. Ceci peut s'expliquer par le fait que les fichiers le plus souvent recherchés sont aussi les fichiers les plus souvent modifiés comme nous le verrons à la section suivante. Donc, en omettant la dimension de modification d'un fichier, nous perdons de la précision par rapport au temps global d'un fichier.

Temps de modification passé par fichier

Nos tâches ne demandant que la modification d'un seul fichier, il est difficile dans ce cas de tirer des conclusions quant à la corrélation entre la pertinence d'un fichier et son temps de modification. On peut cependant émettre l'hypothèse que dans le cas d'un projet avec de nombreux fichiers, un fichier avec un temps de modification important par rapport à un autre

3. Pour établir ces fichiers principaux, nous nous limitons au fichier dont le temps de recherche est supérieur à 5% du temps total de recherche par tâche

avec un temps de modification faible sera un fichier significativement important. Ceci sera discuté à la section 9.2.

8.4.2 Le nombre de revisites comme critère de pertinence

Tout d'abord, définissons la revisite d'un fichier comme étant l'action d'ouvrir un fichier après qu'il ait été fermé ou bien de faire apparaitre de nouveau ce fichier dans l'éditeur après avoir navigué dans un autre.

Les graphiques 8.12, 8.13, 8.14 et 8.15 présentent la moyenne de répartition de revisites par fichier.

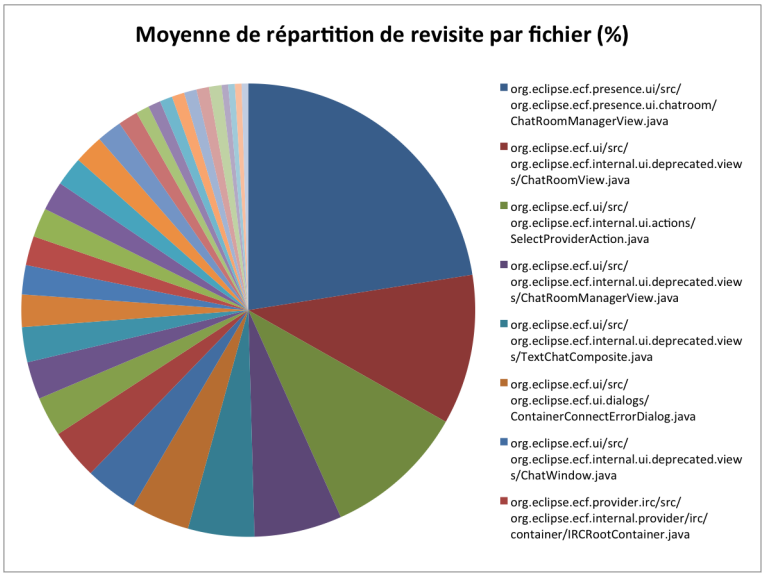


FIGURE 8.12 – Moyenne de répartition de revisites par fichier (%) pour ECF

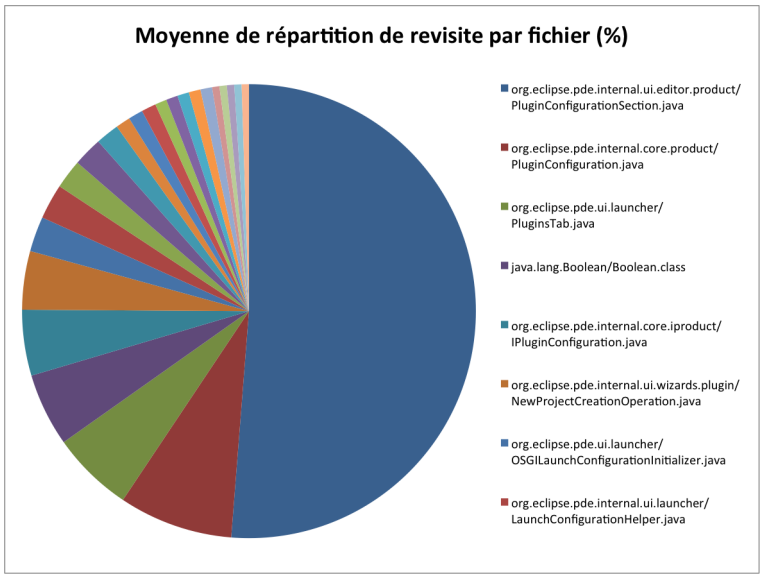


FIGURE 8.13 – Moyenne de répartition de revisites par fichier (%) pour PDE

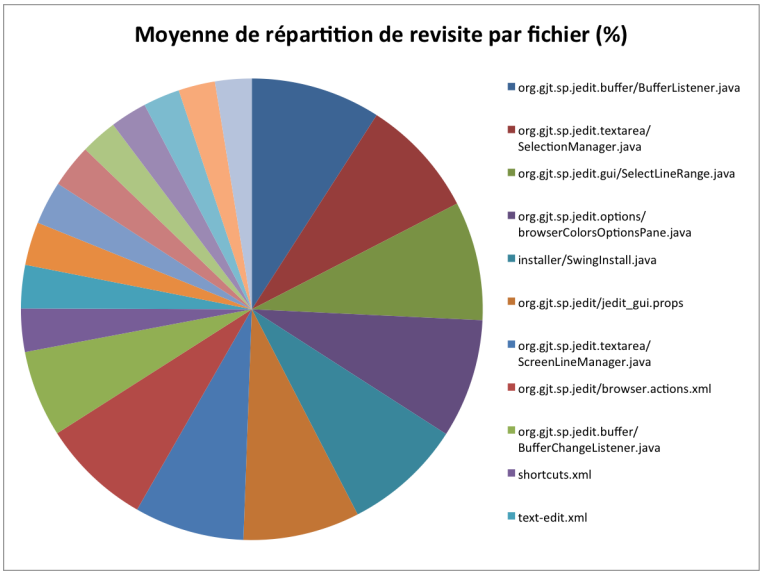


FIGURE 8.14 – Moyenne de répartition de revisites par fichier (%) pour jEdit

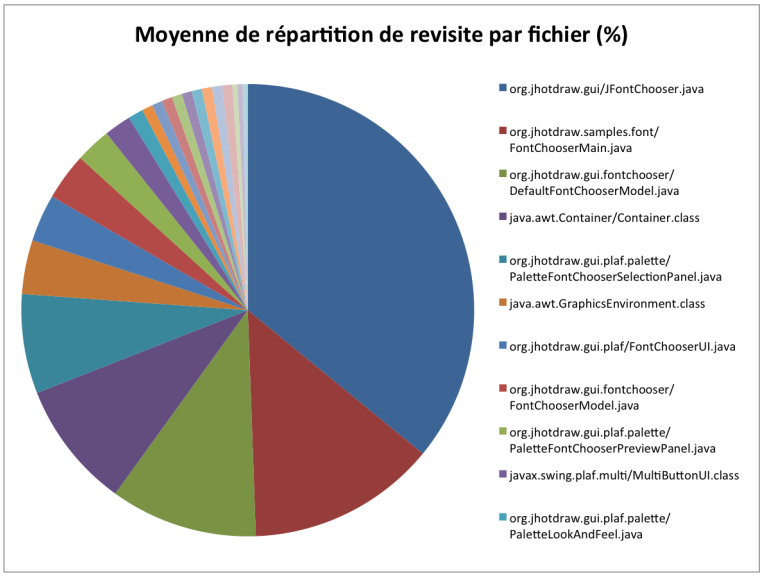


FIGURE 8.15 – Moyenne de répartition de revisites par fichier (%) pour JHotDraw

Bien que les fichiers significativement pertinents apparaissent dans la liste des fichiers ayant le plus grand nombre de revisites, il est moins évident de dire qu'un fichier significativement pertinent aura toujours un nombre de revisites important. Nous pouvons le voir avec jEdit dont la répartition est beaucoup plus variée. Ceci est aussi valable pour les fichiers hautement pertinents.

La seule conclusion que nous pouvons tirer du nombre de revisites par fichier est que la pertinence d'un fichier est en relation avec son nombre de revisites sans pouvoir en distinguer son type de pertinence.

8.4.3 Discussion sur la différentiation des fichiers hautement et significativement pertinents

Au regard de ces critères de pertinence, il est difficile de distinguer les fichiers significativement pertinents des fichiers hautement pertinents. Les résultats obtenus portent à faire croire que l'ensemble des fichiers significativement pertinents est un sous ensemble de l'ensemble des fichiers hautement pertinents. En effet, cette croyance est motivée par le fait qu'un fichier à modifier contient beaucoup d'éléments de compréhension pour effectuer la modification.

8.5 Menaces sur les résultats de l'expérimentation

Nous expliquerons dans cette section l'ensemble des menaces que comporte l'expérimentation. Même si l'expérimentation tend à être contrôlée à tous niveaux, certains facteurs peuvent rendre certains résultats biaisés.

8.5.1 Le choix des sujets

Dans chaque expérimentation, le choix des sujets constitue la plus grande menace. En effet, les résultats dépendent de l'échantillon utilisé pour l'expérimentation. Dans notre expérimentation, notre échantillonnage ne contient qu'un seul sujet qui avait de l'expérience professionnelle et non universitaire. On peut donc dire que notre échantillon de sujets est majoritairement dicté par une pratique universitaire de la programmation. Ceci pourrait donc modifier certains résultats.

De plus, pour que l'échantillonnage soit complet, il faut un nombre suffisant de participants. Notre expérimentation a été effectuée sur une vingtaine de sujets et 15 ont pu être validés. Le défaut de ce genre d'expérimentation est le temps demandé pour effectuer celle-ci et l'obligation de la présence du sujet devant nous pour pouvoir contrôler l'expérimentation. L'échantillon se

réduit donc très vite. Cela prend aussi un temps considérable de pouvoir organiser un planning qui s'adapte aux obligations de chacun.

8.5.2 L'expérience en programmation des sujets

Une des grandes difficultés de l'expérimentation est de connaitre l'expérience des sujets. Pour ce faire, nous avions, lors d'un pré-questionnaire, établi une série de questions pour évaluer ce niveau. Nous avons établi ce niveau grâce au nombre d'années d'expérience du sujet. Certains sujets furent refusés du fait qu'ils avaient une grande expérience théorique du langage mais n'avaient jamais participé à un projet de développement.

Si une trop faible expérience pose problème, une trop haute expérience est aussi un risque. En effet, lors de l'échantillonnage des sujets, nous ne pouvions nous baser uniquement sur ce pré-questionnaire, c'est-à-dire que l'évaluation personnelle de ses capacités par chaque participant. Il s'est avéré que certains sujets jugeaient leur expérience en Java très faible alors qu'ils avaient une bonne connaissance du langage.

8.5.3 Pratique des sujets avec l'environnement expérimental

Notre expérimentation concernait une tâche de programmation à effectuer via l'environnement de développement Eclipse. Celui-ci comporte un ensemble d'outils tels que le débugage, des outils de recherche de contenu. Pour ne pas perturber nos sujets ni nous écarter de la façon habituelle qu'a le sujet d'utiliser Eclipse, nous avons choisi de ne pas interdire ces outils. Ce choix peut être discuté car la connaissance de ces outils permettait de résoudre la tâche plus facilement et plus rapidement. Les interactions pour trouver la solution en sont donc modifiées.

8.5.4 Le choix des tâches à effectuer

Le choix des tâches est aussi important que le choix des sujets. En effet, elles sont la base de toute l'expérimentation. Pour choisir ces tâches, nous avons opté pour une manière rigoureuse et identique comme nous l'avons indiqué à la section 6.2.6. Cependant, cette sélection a entrainé le choix de tâches se limitant à la modification dans un seul fichier et non dans plusieurs. De plus, les tâches choisies sont étalées sur un espace de temps important. Certaines ne sont donc que très rarement observées à l'heure actuelle. Un développeur expérimenté sera donc avantagé par rapport à un jeune développeur.

De plus, la plupart des tâches de maintenance en programmation sont dépendantes d'une technologie particulière (Swing [4], Eclipse Plugin [5]). L'expérience dans l'utilisation de celles-ci va donc jouer un rôle important dans la réussite de la tâche. C'est pourquoi nous avons préféré nous concentrer sur la façon de résoudre la tâche plutôt que sur la modification du code source pour celle-ci. En effet, certains sujets n'ont pas réussi la tâche mais leur résultats restent tout de même utilisables car l'impossibilité pour eux de modifier le code source n'était due qu'à un manque de connaissance de cette technologie particulière.

8.5.5 La motivation du sujet lors de l'expérimentation

Dans une expérimentation telle que la nôtre, la motivation des participants joue un rôle important. En effet, un participant non-motivé pourrait abandonner la tâche ou la réaliser d'une manière totalement aléatoire de telle manière que les données obtenues ne pourront pas être validées. Il est très difficile de juger de cette motivation car une recherche aléatoire peut aussi être due à une personne dont l'expérience en Java est faible. Pour réduire ce risque, nous avons averti les participants du délai l'expérimentation fixé à 60 minutes. Pour certains sujets, leurs données n'ont pu être validées du fait que les personnes, démotivées, ont quitté l'expérimentation après une quinzaine de minutes.

8.5.6 L'interprétation des données

Notre expérimentation récolte des données brutes dont il est impossible d'effectuer un traitement automatique pour en retirer de l'information. Nous devons donc transformer toutes ces données. Cette transformation peut apporter quelques biais dans les résultats. Cependant, dans notre transformation, expliquée à la sous section 7.2.2, nous avons veillé à définir un template d'analyse des vidéos le plus précis possible pour ne perdre aucune information.

4. `http://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html`
5. `http://www.eclipse.org/pde/`

Chapitre 9

Conclusion

Dans un premier temps, ce chapitre liste les différentes parties de ce travail ainsi que les points importants développés au sein de celui-ci. Dans un second temps, nous expliquerons les développements qui peuvent s'inscrire dans la continuité du mémoire.

9.1 Résumé et points importants du mémoire

Dans l'introduction du travail, nous avons présenté un résumé des concepts principaux de ce mémoire : la compréhension logicielle et la maintenance logicielle. Nous avons vu que la maintenance représente une grande partie des coûts d'un logiciel. Avant toute modification d'un fichier, rappelons-le, il faut pouvoir comprendre le projet sur lequel nous travaillons.

Nous avons essayé de voir comment nous pouvions supporter cette maintenance par une aide à la compréhension. Le but de ce travail fut de voir s'il était possible de détecter des fichiers pertinents basés sur la définition de Lee et Kang[37]. Pour établir cette pertinence, nous avons décidé de mettre en place une expérimentation. Ce mémoire résume comment nous avons établi cette expérimentation sur base du livre *"Experimentation In Software Engineering : An Introduction"*[7] ainsi que les résultats que nous avons obtenus.

La première partie de ce mémoire, les chapitres 2 et 3, fait l'état de l'art respectivement des concepts de la compréhension logicielle et de la maintenance logicielle. Nous avons résumé les deux types d'analyse que nous pouvons faire d'un programme : l'analyse statique qui révèle les propriétés d'un programme sans exécution et l'analyse dynamique qui révèle les propriétés d'un programme en cours d'exécution. Nous avons aussi donné la définition de la maintenance logicielle et de ses différents types ainsi que de sa visualisation via plusieurs modèles.

La deuxième partie de ce mémoire a deux buts : donner des rappels théoriques et expliquer la mise en contexte de notre expérimentation. Un expérimentation est composée de 5 étapes principales : la définition, la planification, la phase opérationnelle, la phase d'analyse et interprétation, la phase de présentation que constitue ce mémoire.

Le chapitre 5 définit notre expérimentation. Elle veut analyser la résolution d'une tâche de maintenance. Sa finalité est de catégoriser les fichiers en fonction de leur pertinence vis à vis de leur tâche du point de vue du développeur dans le contexte d'étudiants et de professionnels sur des projets de tailles et complexités différentes.

Le chapitre 6 montre les moyens mis en place pour réaliser l'expérimentation. Nous avons choisi d'effectuer celle-ci sur les projets java ECF, PDE, jEdit et JHotDraw avec des sujets dont la majorité était issue du milieu universitaire. Nous avons ensuite défini une tâche par projet. Chaque tâche était liée à une série de questionnaires pour nous permettre de valider nos résultats. Pour notre expérimentation, nous avons choisi de mettre tout le code source à disposition des sujets sur l'IDE Eclipse pour ensuite étudier leur exploration avec l'hypothèse de pouvoir déceler certains fichiers pertinents.

Dans le même chapitre, nous avons aussi défini l'hypothèse nulle suivante : il est impossible de détecter les fichiers pertinents pour une tâche donnée. Ce qui fait que si nous pouvons rejeter cette hypothèse, il est alors possible de déceler des fichiers pertinents pour une tâche donnée.

Le chapitre 7 explique la phase opérationnelle de l'expérimentation. L'étape importante de cette phase est de définir la façon dont nous allons collecter et transcrire les données des sujets. Nous avons choisi la capture vidéo et de manuellement transcrire les traces vidéos en terme d'interaction où nous notions les temps, les actions ainsi que l'origine de l'interaction.

Enfin dans le chapitre 8, nous avons traité les données contenues dans les interactions pour pouvoir les présenter et ainsi les analyser. Plusieurs résultats ont été trouvés et confirmés.

Nous avons ainsi confirmé que la compréhension d'un logiciel est essentielle et que le temps passé à comprendre les artéfacts du système est plus important que le temps nécessaire à modifier, réellement, un fichier. Il y a donc une perte de temps que l'on doit essayer de réduire.

Nous avons ensuite défini deux critères comme étant des critères de pertinence : le temps par fichier et le nombre de revisites par fichier.

En effet, l'ensemble des données récoltées a permis de définir le temps comme critère de pertinence. Nous avons distingué plusieurs types de temps :

Temps total Nous avons observé que le temps total permet de donner des indications quant à la pertinence du fichier tant au niveau de la compréhension qu'au niveau de la modification.

Temps de recherche Nous avons observé qu'utiliser uniquement le critère de temps de recherche sur un fichier diminuait la précision en comparaison au temps total passé dans le même fichier.

Temps de modification Nous avons observé qu'il n'était pas possible au vu de notre expérimentation de tirer des conclusions sur le temps de modification comme facteur de pertinence.

Le nombre de revisites s'avère être aussi un facteur de pertinence. Cependant, le nombre de revisites n'indique pas la différence entre un fichier hautement pertinent et un fichier significativement pertinent.

Au vu de ces résultats, nous pouvons rejeter notre hypothèse nulle et dire qu'il est possible de déceler les fichiers importants.

Toutes ces conclusions sont mises en doute par les menaces vues à la section 8.5. Cependant, même si ces menaces semblent importantes, elles n'invalident pas l'expérimentation.

9.2 Travaux futurs

Cette section explique ce qui pourrait être fait pour continuer ce travail ou l'améliorer. Nous proposerons aussi la réutilisation de l'expérimentation à d'autres fins.

9.2.1 Point de vue expérimentation

Le point faible de toute expérimentation est d'évaluer si les résultats obtenus sont bien représentatifs de la réalité ou si ils ne concernent que les sujets et tâches définies par les examinateurs. On pourrait donc imaginer répliquer cette expérience en faisant varier plusieurs critères qui représentent les menaces les plus importantes de notre expérimentation.

La population Faire varier la population et le type de sujets pour vérifier si ces résultats sont applicables à des sujets ayant une plus grande expérience professionnelle. Il serait aussi possible de confronter les étudiants d'autres universités aux tâches que nos sujets ont résolues pour analyser les différentes approches de la programmation que les universités enseignent.

Les tâches Reproduire l'expérience avec des tâches différentes.

Les projets dans la durée Récolter les données sur un projet dans une plus longue durée pour analyser les variations de la pertinence des fichiers sur le long terme.

9.2.2 Point de vue logiciel et outil

Les résultats étant concluants, il est intéressant de créer un moyen de les visualiser dans l'IDE Eclipse. Il faudrait, dans un premier temps, analyser les actions réellement nécessaires pour ensuite les capturer. L'étape d'analyse étant facilement automatisable, il suffirait de la meilleure façon de visualiser ces résultats dans un outil annexe à l'IDE Eclipse.

Les étapes de capture manuelle et d'analyse manuelle de traces expliquées au chapitre 7 et 8 devraient être automatisées. Notre approche représentée par la figure 9.1 vue au chapitre 1 comporterait l'étape (1) et (3) automatisées. On peut imaginer une façade de visualisation entre l'*input* des fichiers pertinents et l'*output* des fichiers pertinents.

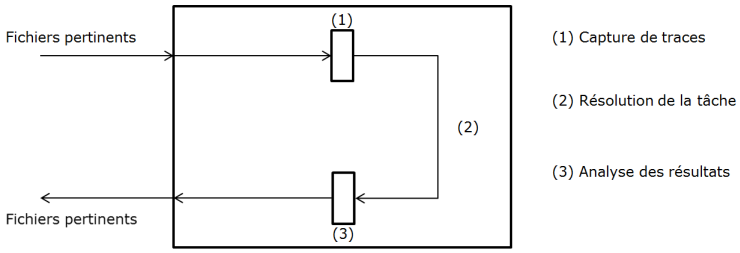


FIGURE 9.1 – Itération d'une maintenance

9.2.3 Réutilisation de l'expérimentation

Pour obtenir les informations dont nous avions besoin, nous avons défini des interactions résumées par la table 7.1 à la section 7.2.2. Certains de ces concepts concernent l'exploration de programme et la compréhension d'un fichier par un développeur. Dans notre cas, ce développeur trouve la solution à une tâche sans connaitre le projet sur lequel il travaille. Il pourrait être intéressant d'utiliser les données de notre expérimentation pour vérifier certains modèles d'exploration de programme.

Bibliographie

[1] Armstrong A., Takang A.A., and Grubb P.A. *Software Maintenance : Concepts and Practice*. International Thomson Computer Press, 1996.

[2] Ko A.J., Myers B.A., Coblenz M.J., and Aung H.H. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions On Software Engineering*, 32(12) :971–987, december 2006.

[3] Cornelissen B., Zaidman A., van Deursen A., Moonen L., and Koschke R. A systematic survey of program comprehension through dynamic analysis. *IEEE Trans. Software Eng.*, 2009.

[4] Lientz B.P. and Swanson E.B. *Software Maintenance Management*. Addison-Wesley Publishing Co., 1980.

[5] Boehm B.W. A spiral model of software development and enhancement. *IEEE Computer*, 21(5) :61–72, May 1988.

[6] Robson C. *Real World Research : A Ressource for Social Scientists and Practitioner-Researchers*. Blackwell Publishing, 2002.

[7] Wohlin C., Runeson P, Höst M., Ohlsson M.C., Regnell B., and Wesslén A. *Experimentation In Software Engineering : An Introduction*. Kluwer Academic Publishers, 2000.

[8] Manning C.D., Raghavan P., and Schutze H. Introduction to information retrieval. *Cambridge University Press*, 1, 2008.

[9] Blei D.M., Ng A.Y., and Jordan M.I. Latent dirichlet allocation. *the Journal of machine Learning research*, 3 :993–1022, 2003.

[10] Abdulrazzak E.Z. and Ghani I. Secure software design maintenance using enhanced task-oriented security maintenance model. 2013.

[11] Salton G., Wong A., and Yang C.S. A vector space model for automatic indexing. *Communications of the ACM*, 18(11) :613–620, 1975.

[12] Müller H. and Villegas N. *Runtime Evolution of Highly Dynamic Software*, chapter 8. 2014.

[13] Jacobson I. *Object oriented software engineering : a use case driven approach*. 1992.

[14] Sommerville I. *Software Engineering*. Addison-Wesley Publishing Co., 2011.

[15] IEEE. *A Guide to the Project Management Body of Knowledge*. IEEE, 2004.

[16] IEEE. *Software Engineering — Software Life Cycle Processes — Maintenance*. IEEE, 2006.

[17] ISO. *Quality management systems - Guidelines for quality management in projects*. ISO, 2003.

[18] Nielsen J. *Usability Engineering*. Morgan Kaufmann Publishers Inc., 1993.

[19] Sillito J., Murphy G.C., and De Volder K. Asking and answering questions during a programming change task. *IEEE Transactions On Software Engineering*, 34(4) :434–451, juillet/aout 2008.

[20] Singer J., Elves R., and Storey M. Navtracks : supporting navigation in software maintenance. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 325–334. IEEE, Sept 2005.

[21] Hainaut J.-L. *Bases de données : Concepts, utilisation et développement*. Dunod, 2009.

[22] Maletic J.I. *The Software Service Bay : A Knowledge Based Software maintenance methodology*. PhD thesis, Wayne State University, 1995.

[23] Aggarwal K. K. and Senghi Y. *Software Engineering*, chapter 9. New Age International Publishers, 2007.

[24] Petit M. Cours de gestion de projet informatiques. 2012.

[25] Lehman M.M. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9) :1060–1076, Sept 1980.

[26] Robillard M.P., Walker R.J., and Zimmermann T. Recommendation systems for software engineering. *IEEE Software*, 27(4) :80–86, July 2010.

[27] Chapin N., Hale J.E., Khan K.M., Ramil J.F., and Tan W.-G. Types of software evolution and software maintenance. *Journal of Software Maintenance and Evolution : Research and Practice*, 2001.

[28] Noughi N. Understanding the data manipulation behavior of data-intensive systems. 2014.

[29] Schneidewind N.F. The state of software maintenance. *Software Engineering, IEEE Transactions on*, SE-13(3) :303–310, March 1987.

[30] Caserta P. *Analyse statique et dynamique de code et visualisation des logiciels via la métaphore de la ville : contribution à l'aide à la compréhension des programmes*. PhD thesis, Université de Lorraine, 2012.

[31] Chen P. The entity-relationship model-toward a unified view of data. *ACM Transactions on Database Systems*, 1(1) :9–36, March 1976.

[32] Klint P. Introduction to software evolution. http://fr.slideshare.net/devnology/introduction-to-software-evolution-the-software-volcano, 2011. En ligne, accédé le 5 juillet 2014.

[33] Thiran P. Cours d'analyse et de modélisation des systèmes d'information. 2012.

[34] Betala P.K., Jhamad, and S. Kumaresh S. Entretian model for software maintenance. *International Journal of Computer Science & Communication Networks*, 3(4) :276–283, 2013.

[35] Madsen R., Sigurdsson S., Hansen L., and Larsen J. Pruning the vocabulary for better context recognition. *Int'l Conf. Pattern Recognition*, pages 483–488, 2004.

[36] Deerwester S., Dumais S.T., Furnas G.W., and Landauer T.K. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6) :391, September 1990.

[37] Lee S. and Kang S. Clustering navigation sequences to create contexts for guiding code navigation. *The Journal of Systems and Software*, 86(8) :2154–2165, august 2013.

[38] Thomas S.W., Hassan A.E., and Blostein D. *Mining Unstructured Software Repositories*, chapter 5. 2014.

[39] Biggerstaff T. and Richter C. Reusability framework, assessment, and directions. *IEEE Software*, 4(2) :41–49, March 1987.

[40] Fritz T. and Murphy G.C. Determining relevancy : How software developers determine relevant information in feeds. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 1827–1830. ACM, 2011.

[41] Basili V.R. Viewing maintenance as reuse oriented software developement. *IEEE Software*, pages 19–25, January 1990.

[42] Osborne W.M. and Chikofsky E.J. Fitting pieces to the maintenance puzzle. *IEEE Computer*, January 1990.

Annexes

A Formulaires

A.1 Formulaire projet Eclipse

Subject: _____

Pre-experiment questionnaire

What is your gender?

- ☐ Male
- ☐ Female

What is your study level?

- ☐ Bachelor
- ☐ Master
- ☐ Ph. D
- ☐ Post-Doc
- ☐ Professor
- ☐ Other :_____

What is your proficiency in English?

(1 = very bad and 5 = very good)

☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5

How many year(s) have you been programming in Java?

What is your experience/knowledge of Java?

(1 = very bad and 5 = very good)

☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5

Do you use the Eclipse IDE to develop Java program?

- ☐ Yes
- ☐ No

How many years have you been using the Eclipse IDE?

What is your experience/knowledge of the Eclipse IDE?

(1 = very bad and 5 = very good)

☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5

Have you already developed an Eclipse plugin with the Eclipse IDE?

- ❑ Yes
- ❑ No

What do you know about the architecture of an Eclipse plugin?

_____

_____

What is your knowledge of plugin development?

(1 = very bad and 5 = very good)

❑ 1 ❑ 2 ❑ 3 ❑ 4 ❑ 5

What is your knowledge of framework development?

(1 = very bad and 5 = very good)

❑ 1 ❑ 2 ❑ 3 ❑ 4 ❑ 5

A.2 Formulaire projet Java simple

Subject: _____

Pre-experiment questionnaire

What is your gender?

- ❑ Male
- ❑ Female

What is your study level?

- ❑ Bachelor
- ❑ Master
- ❑ Ph. D
- ❑ Post-Doc
- ❑ Professor
- ❑ Other : _____

What is your proficiency in English?

(1 = very bad and 5 = very good)

❑ 1 ❑ 2 ❑ 3 ❑ 4 ❑ 5

How many year(s) have you been programming in Java?

What is your experience/knowledge of Java?

(1 = very bad and 5 = very good)

❑ 1 ❑ 2 ❑ 3 ❑ 4 ❑ 5

Do you use the Eclipse IDE to develop Java program?

- ❑ Yes
- ❑ No

How many years have you been using the Eclipse IDE?

What is your experience/knowledge of the Eclipse IDE?

(1 = very bad and 5 = very good)

❑ 1 ❑ 2 ❑ 3 ❑ 4 ❑ 5

B Résumé des projets

B.1 ECF

ECF

Description:

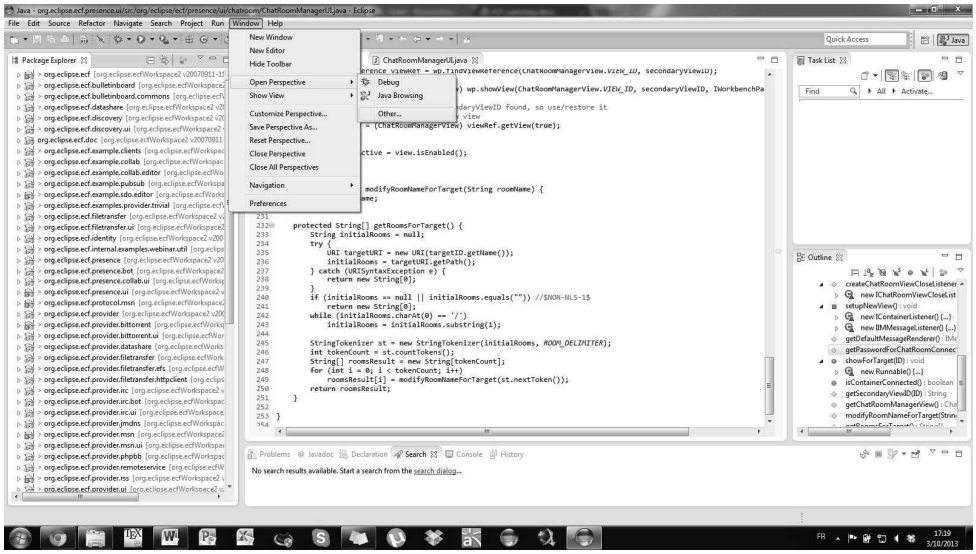
“Eclipse Communication Framework”, aka ECF, is a framework for building distributed servers, applications, and tools. It can also be viewed as an Eclipse plugin to make communications between people easy.

For example, when developers work, they need to talk to other developers without switching from their Eclipse IDE to chat or talk. ECF allows developers to communicate through their Eclipse IDE.

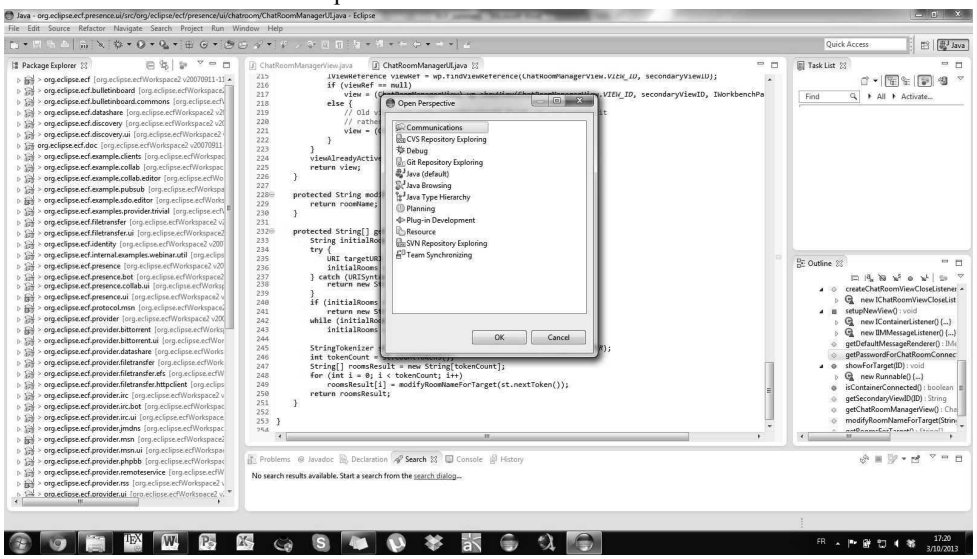
Example:

For example, if you want to connect to IRC (Internet Relay Chat) with ECF, then you can follow those steps:

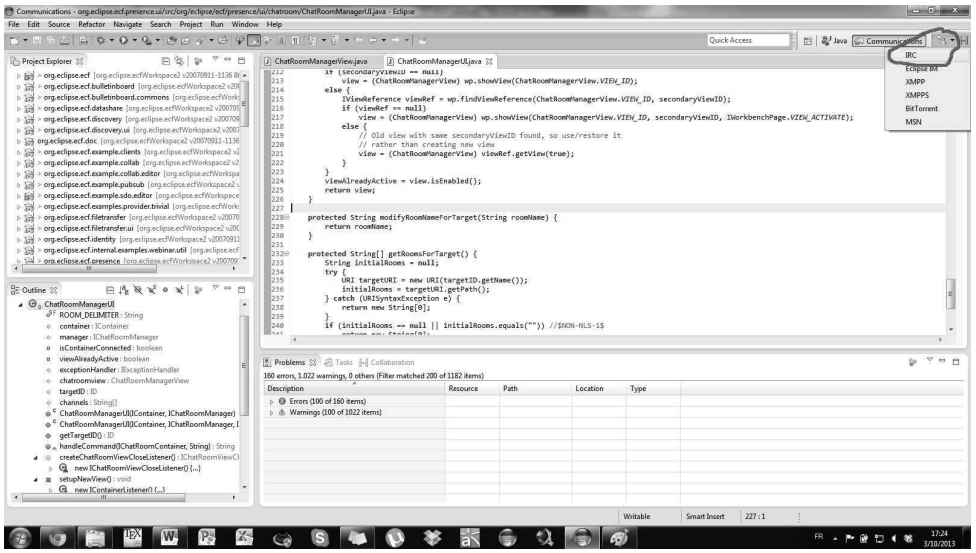
1. Open communication perspective. “Window ->Open Perspective ->Other”



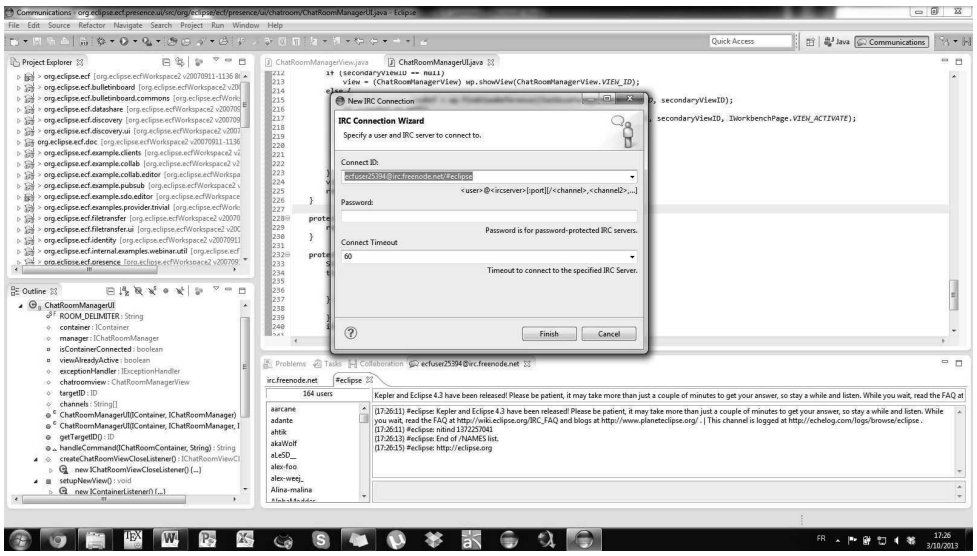
2. Choose “Communications” and press “*OK”*



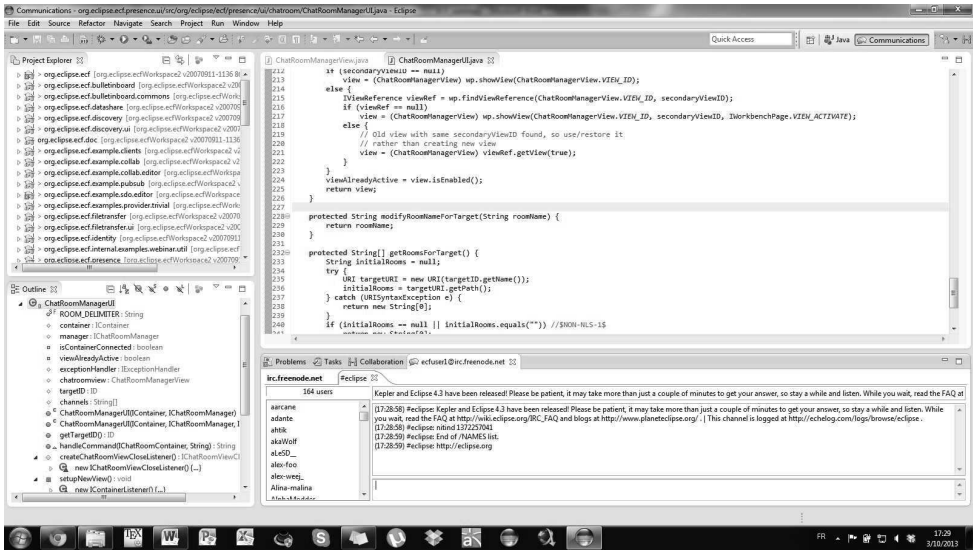
3. Click on the arrow and choose “IRC”



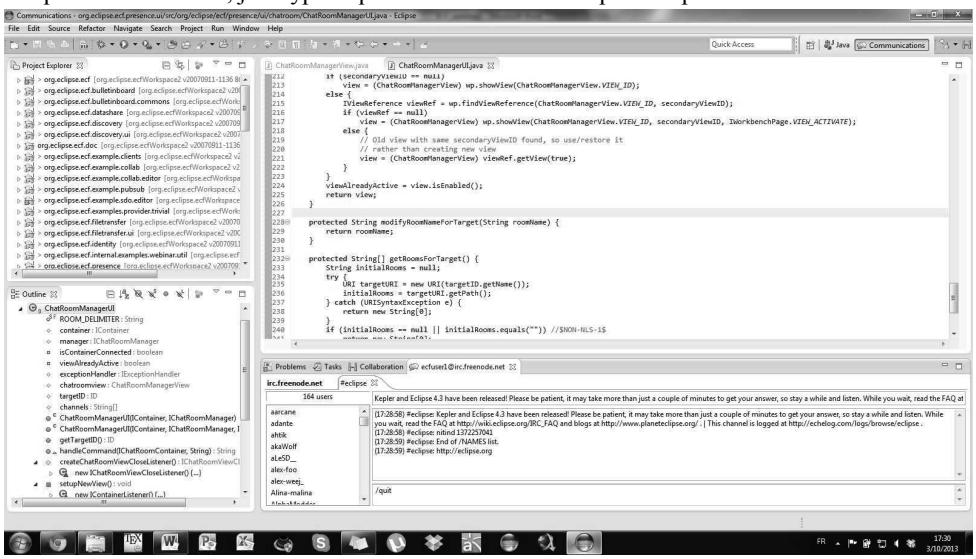
4. Enter your pseudo, choose your channel (something like #name_channel) and press “Finish”



5. You are now connected to the channel and can communicate with people



6. To quit the IRC client, just type “/quit” into the channel input and press “Enter”



B.2 PDE

PDE

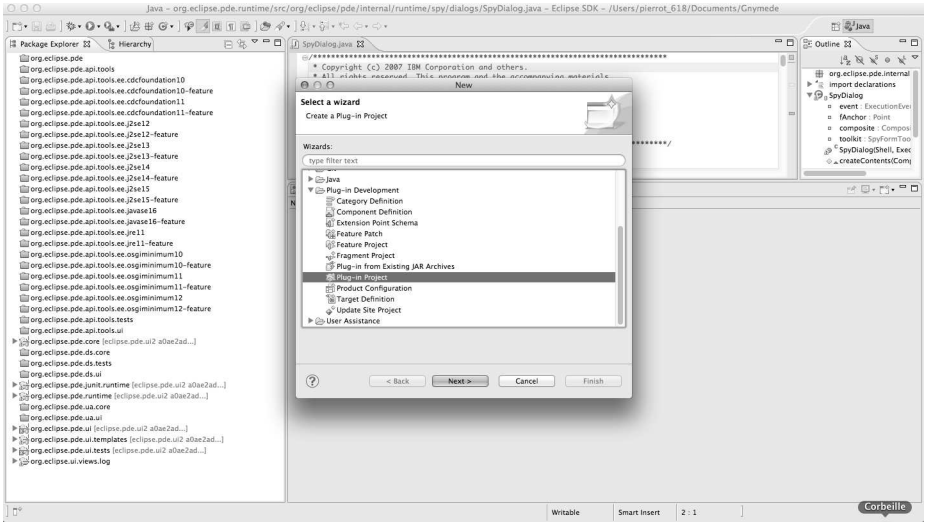
Description:

Eclipse provides a tool (plug-in) called *Plug-in Development Environment* (PDE) that helps developers to create, develop, test, debug, build, and deploy Eclipse plug-ins.

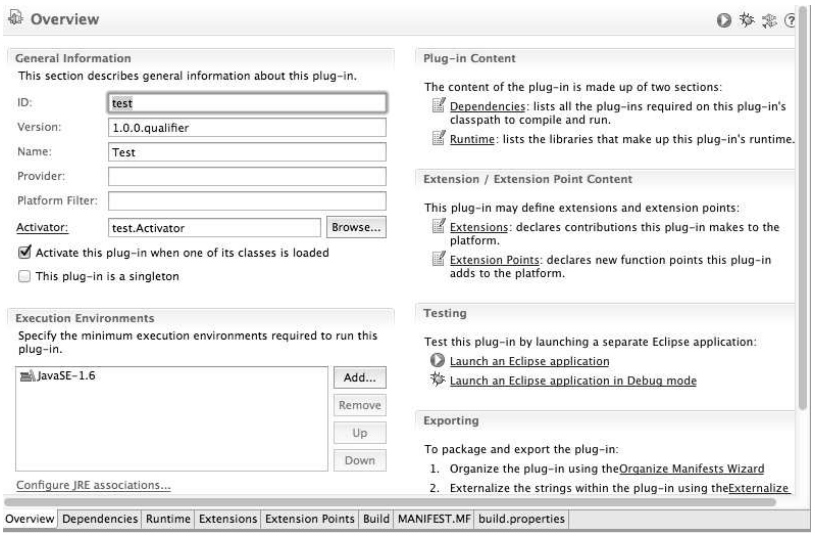
Example:

With PDE, developers have a user interface to create plugins and to specify their configurations.

1. Click on "New -> other -> Plug in project". You can also see that PDE allows you to create other features related to plugins.



2. Here you can see that PDE gives you a user interface to specify dependencies or the Java environment without going into the manifest file. It helps avoid specifications mistakes.



B.3 jEdit

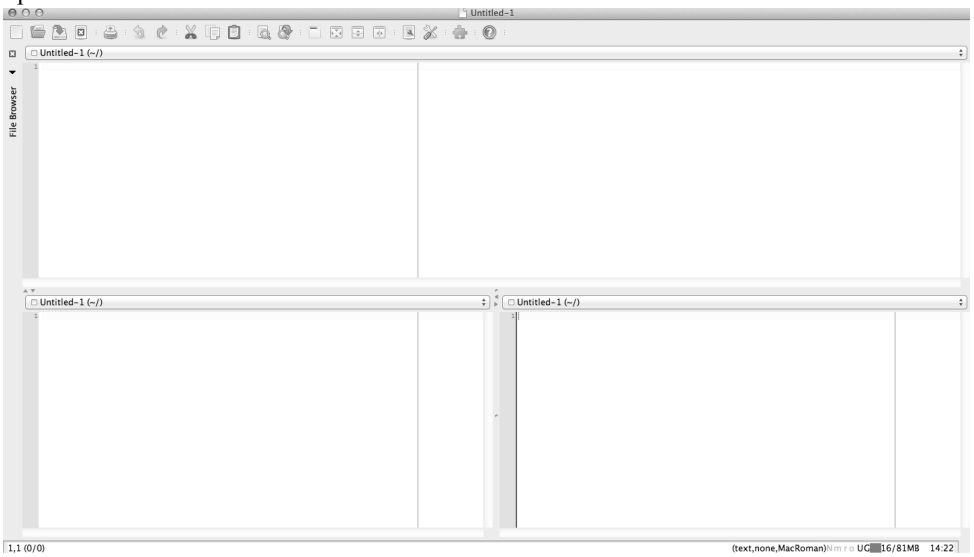
jEdit

Description:

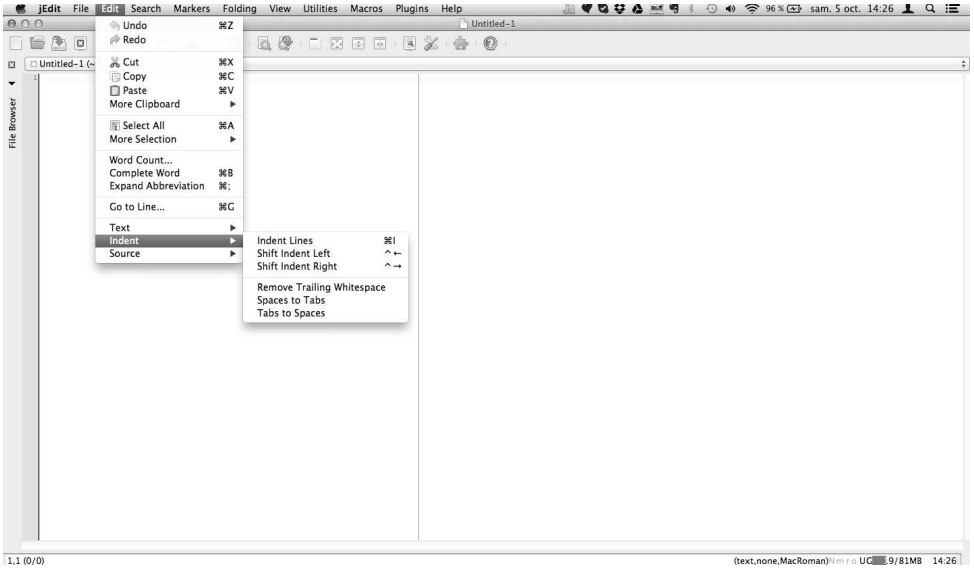
jEdit is a mature developers' text editor with hundreds of person-years of development behind it (counting the time of developing plugins). It allows developers to write code in multiple languages and to define some editor configurations (indentation, colors and--or splitted views).

Example:

Here are screenshots of some features
Splitted windows:



Indentation features:



B.4 JHotDraw

JHotDraw

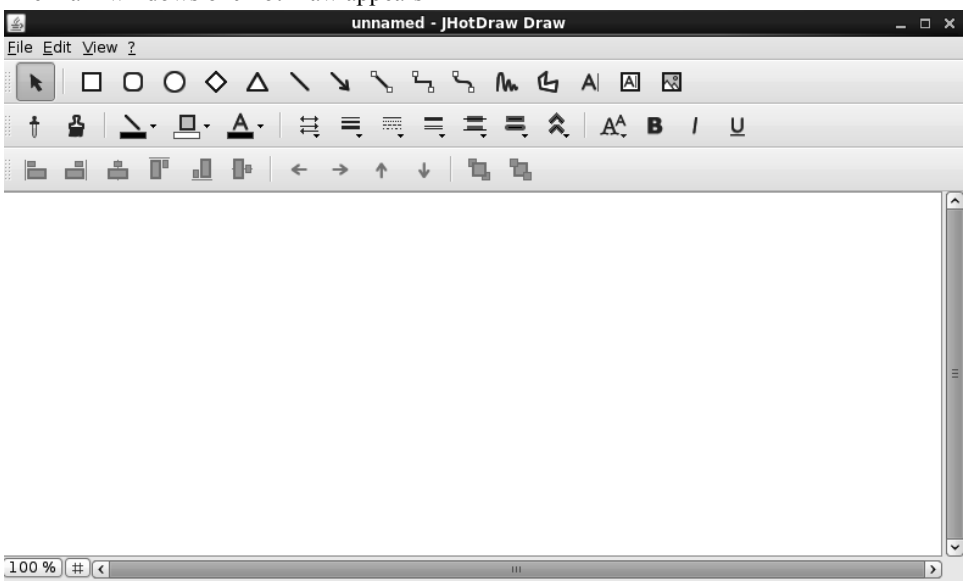
Description

JHotDraw is a Java GUI (Graphical User Interface) framework for technical and structured graphics. It can be used to build animations, figures, drawings and many others things with this tool.

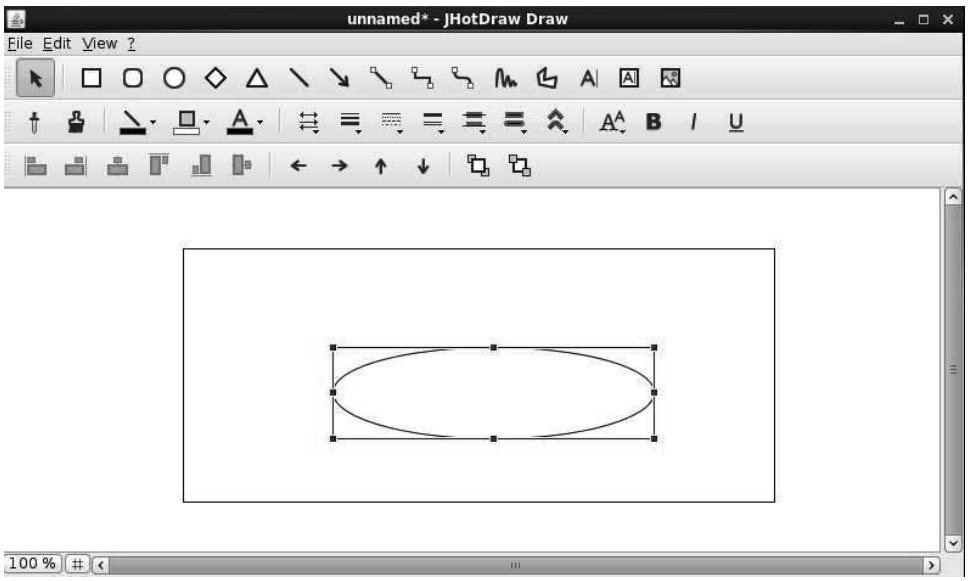
Example

For example, if you want to create a figure with JHotDraw, you could follow these steps:

1. Open a terminal and type “cd Desktop/”. Press “Enter” and type “java –jar JHotDraw_Draw.jar”.
2. The main windows of JHotDraw appears



3. You can now draw a figure. For this example, I will make a rectangle with a circle inside it.



C Présentation des tâches

C.1 ECF

Task 2: Presentation

Task name

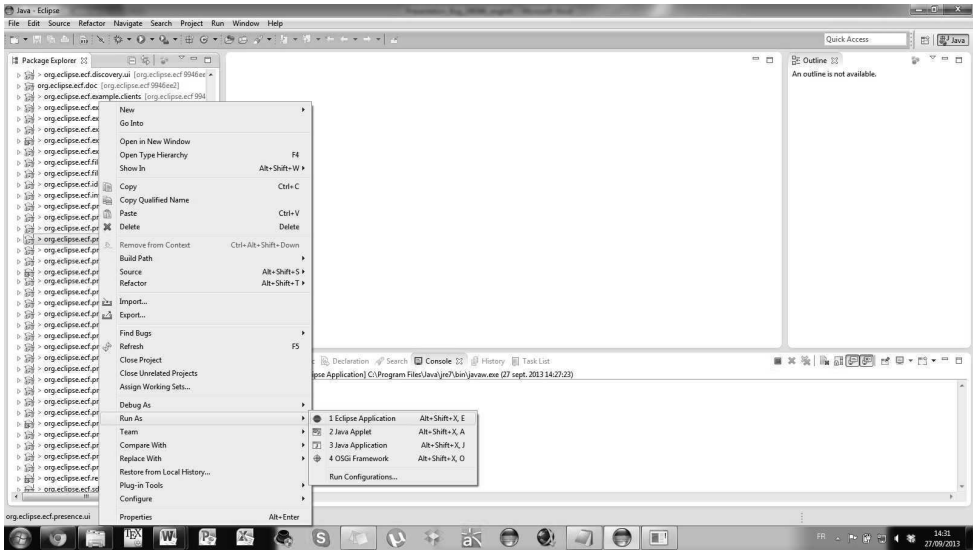
IRC channel output of Copy/Select All does not work correctly.

Description

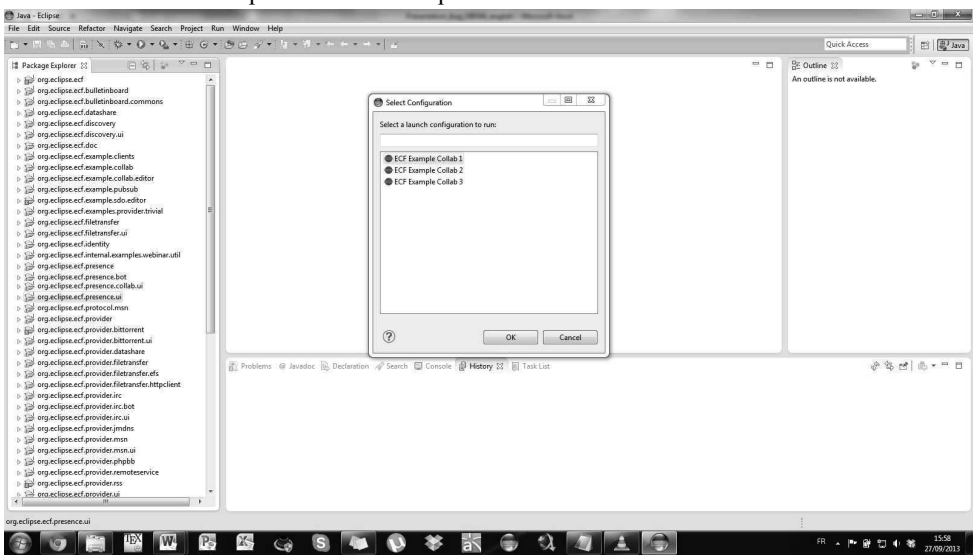
Enter IRC channel (e.g., #eclipse-dev).
Right click on the chat output text area to bring up Copy/Clear/Select All menu.
Choose Select All.
Expected: the channel's text is selected.
Actual: root container's output (irc.freenode.net) is selected rather than the channel's output.

Steps-to-reproduce

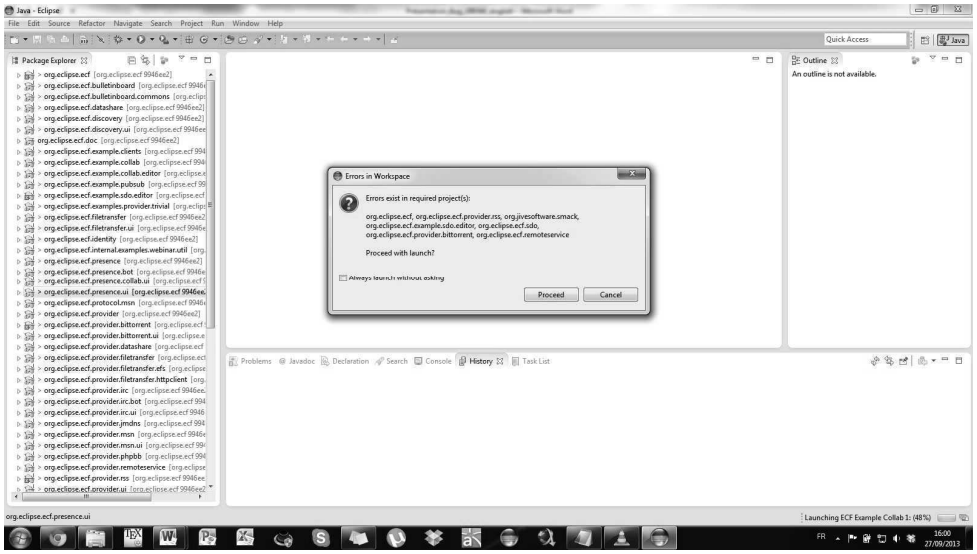
1. Call the experimenter.
2. Run the “org.eclipse.ecf.presence.ui” project as an Eclipse application.



3. Choose the “ECF Example Collab 2*”* and press “OK”.

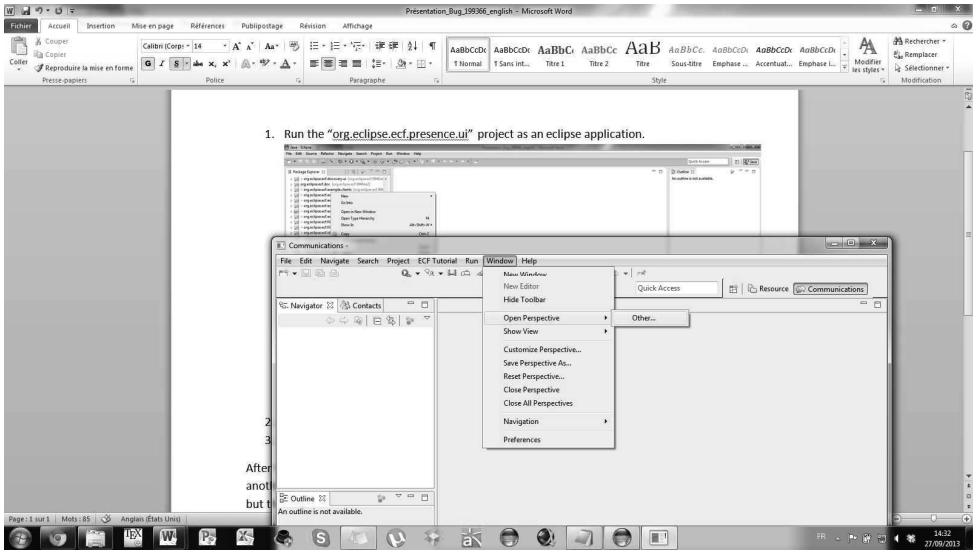


4. Ignore the error message by clicking on “Proceed*”*.

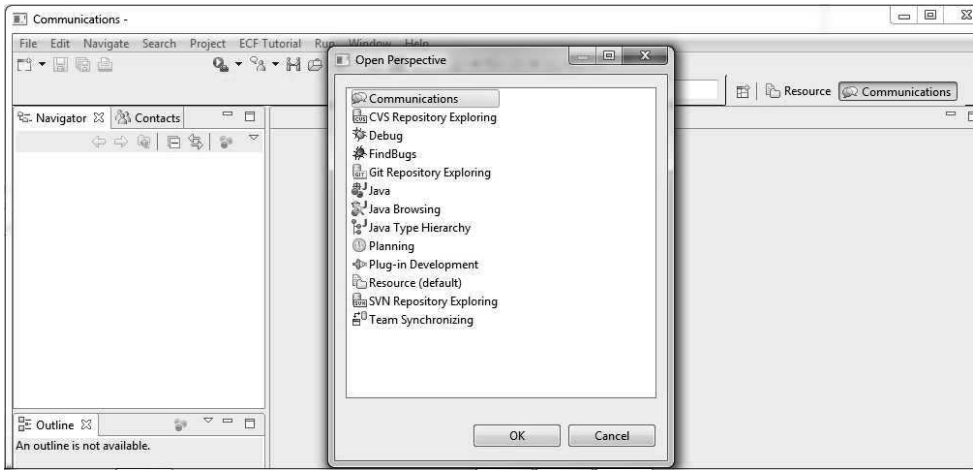


5. Wait some time to see an Eclipse window appear.

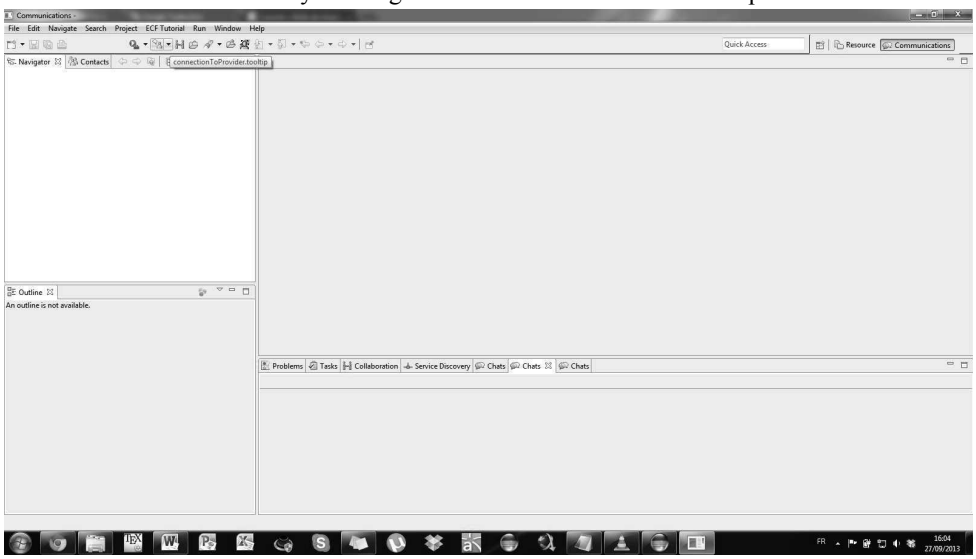
6. Open the Communications perspective by clicking on “Window -> Open Perspective -> Other…”



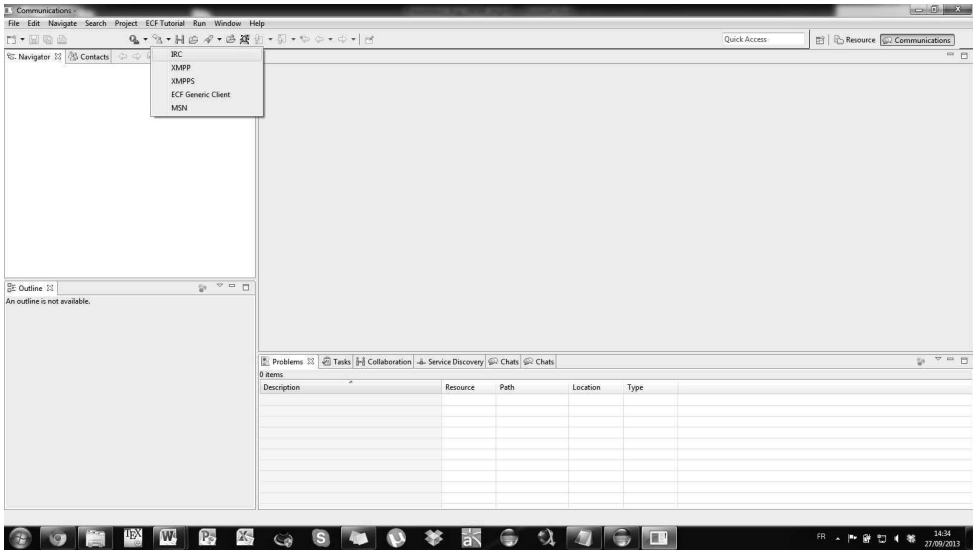
7. Choose “Communications” and press “OK”.



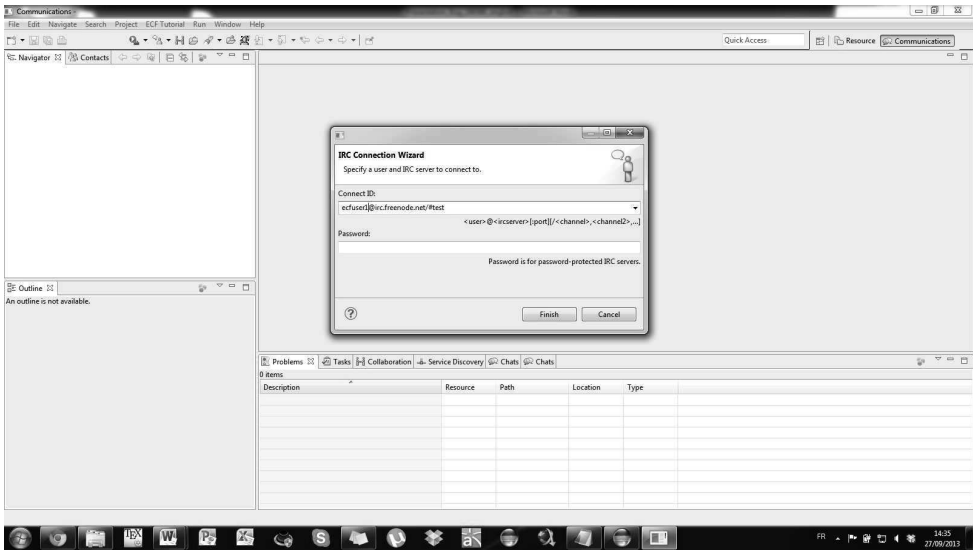
8. Connect to an IRC channel by clicking on the bottom arrow next to the “person”.



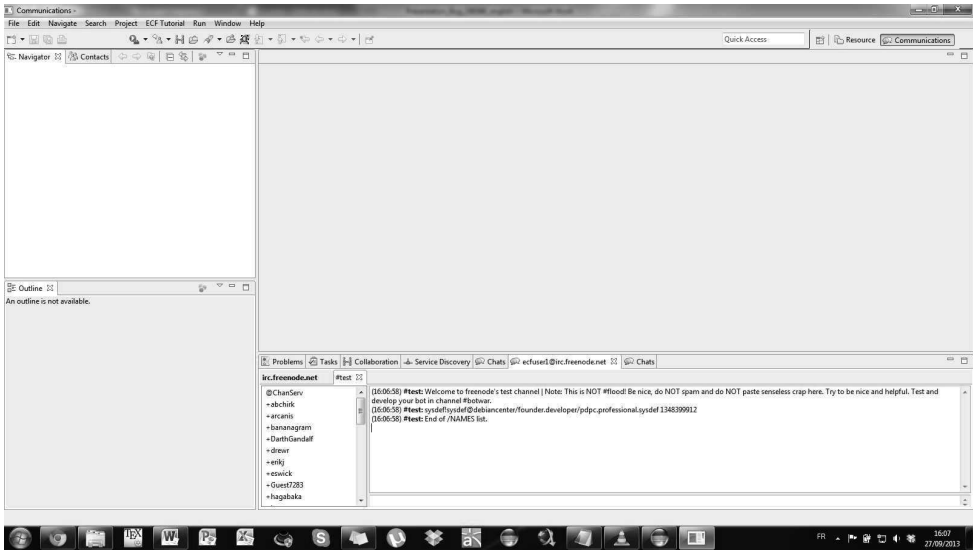
9. Press “IRC”.



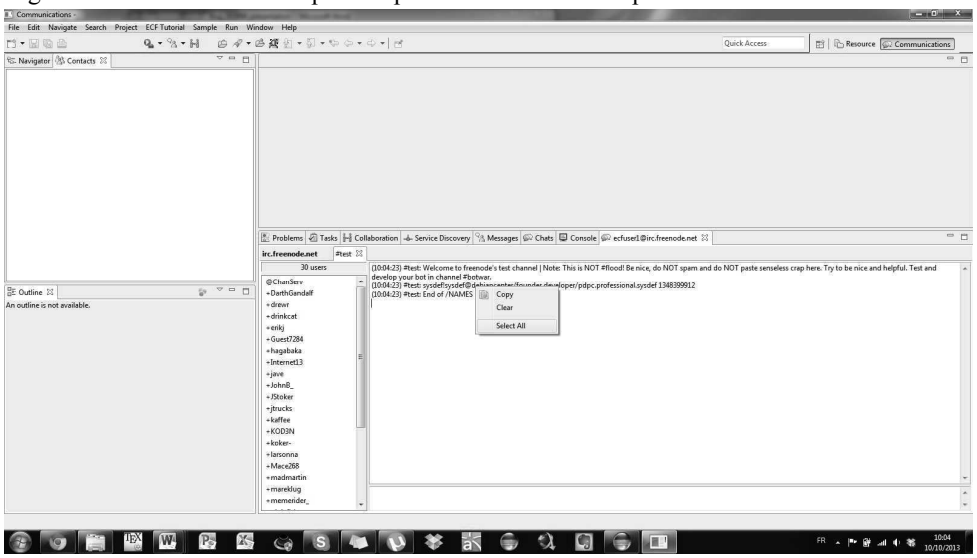
10. Enter the pseudo “ecfuser1” and choose the channel #test on irc.freenode.net. Press “Finish”.



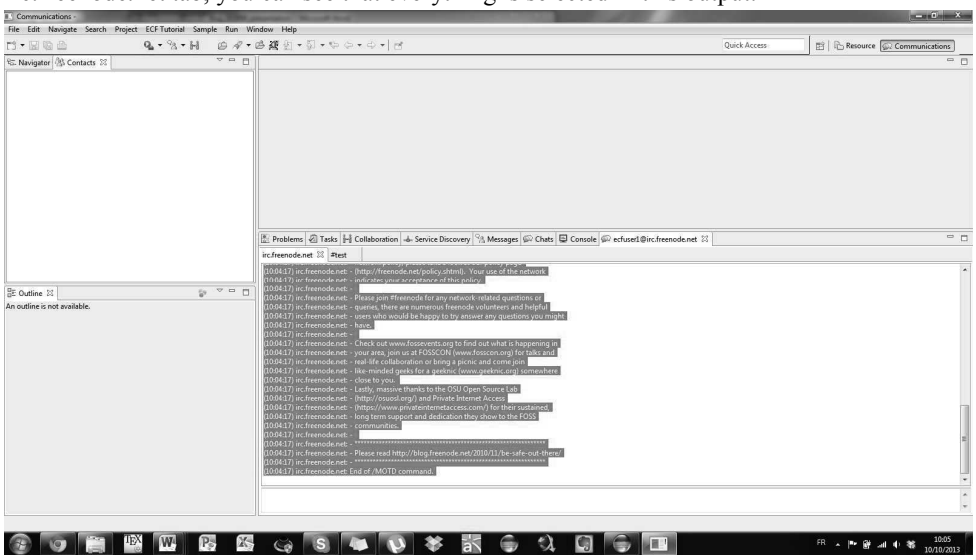
11. Now, you are connected to the IRC channel #test on irc.freenode.test.



12. Right click on the channel output and press the “Select All” option.



13. As you can see, nothing is selected on the #test channel but if you go to the irc.freenode.net tab, you can see that everything is selected in this output.



14. Quit the launched Eclipse and go back to the original Eclipse.

C.2 PDE

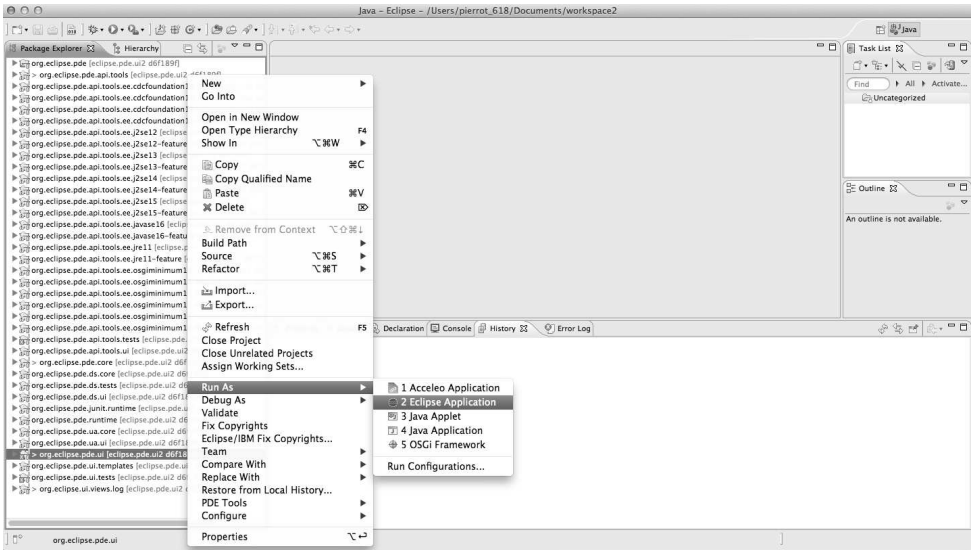
Task 4: Presentation

Task name
Autostart values are not persisted correctly in the product file

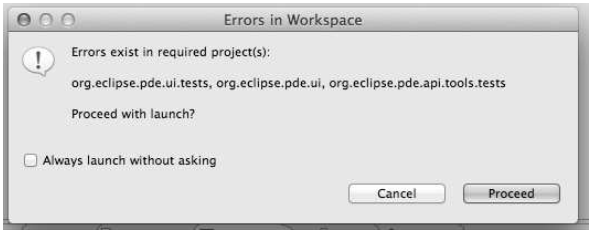
Description
On the Configuration page of the product editor, add a plug-in and set autostart to 'true'.
Save the file.
Open the file in a text editor, and see how the value of the 'autostart' attribute is still set to false.

Steps-to-reproduce

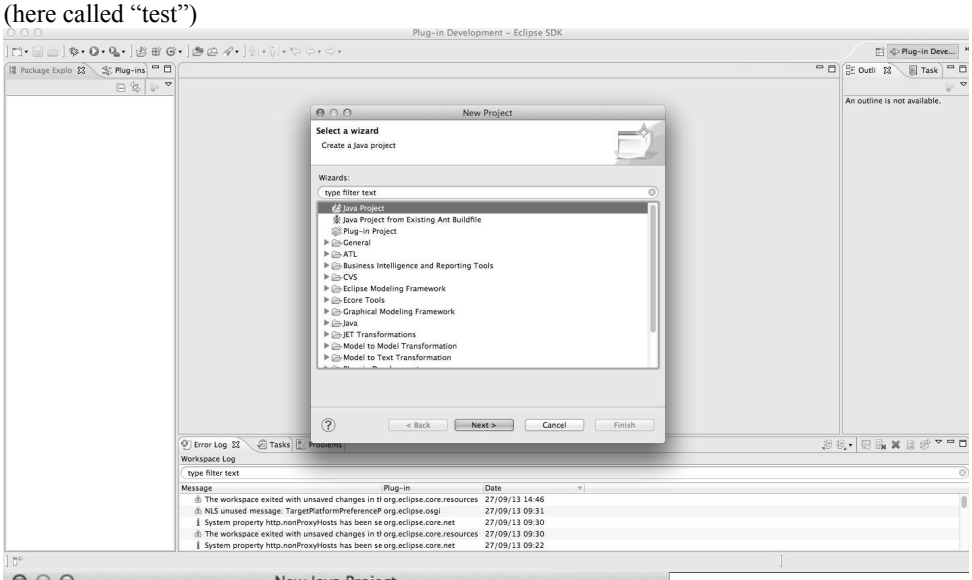
1. Call the experimenter.
2. Run the “org.eclipse.pde.ui” project as an eclipse application.

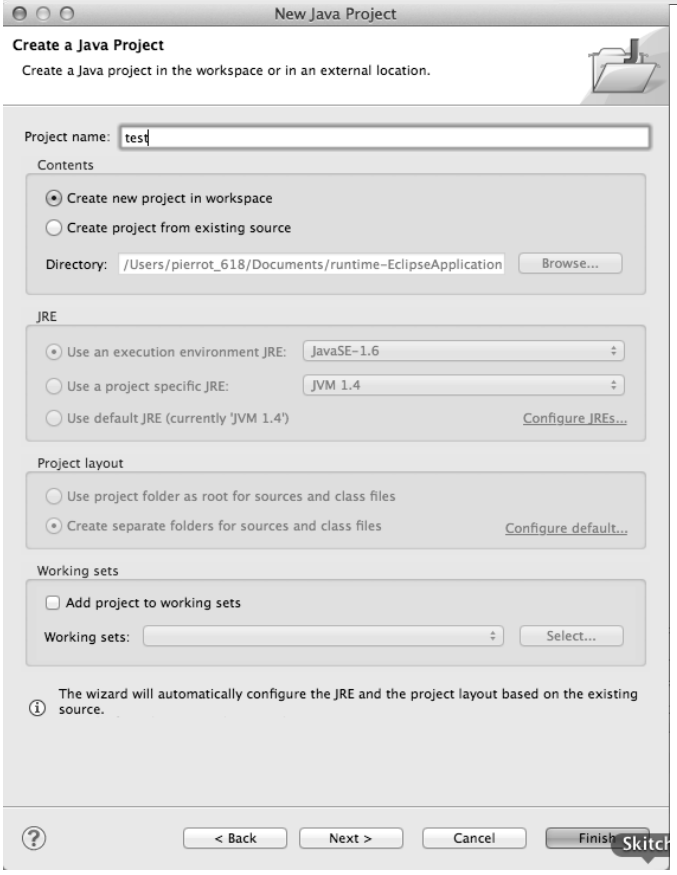


3. Ignore error message and click on “Proceed”

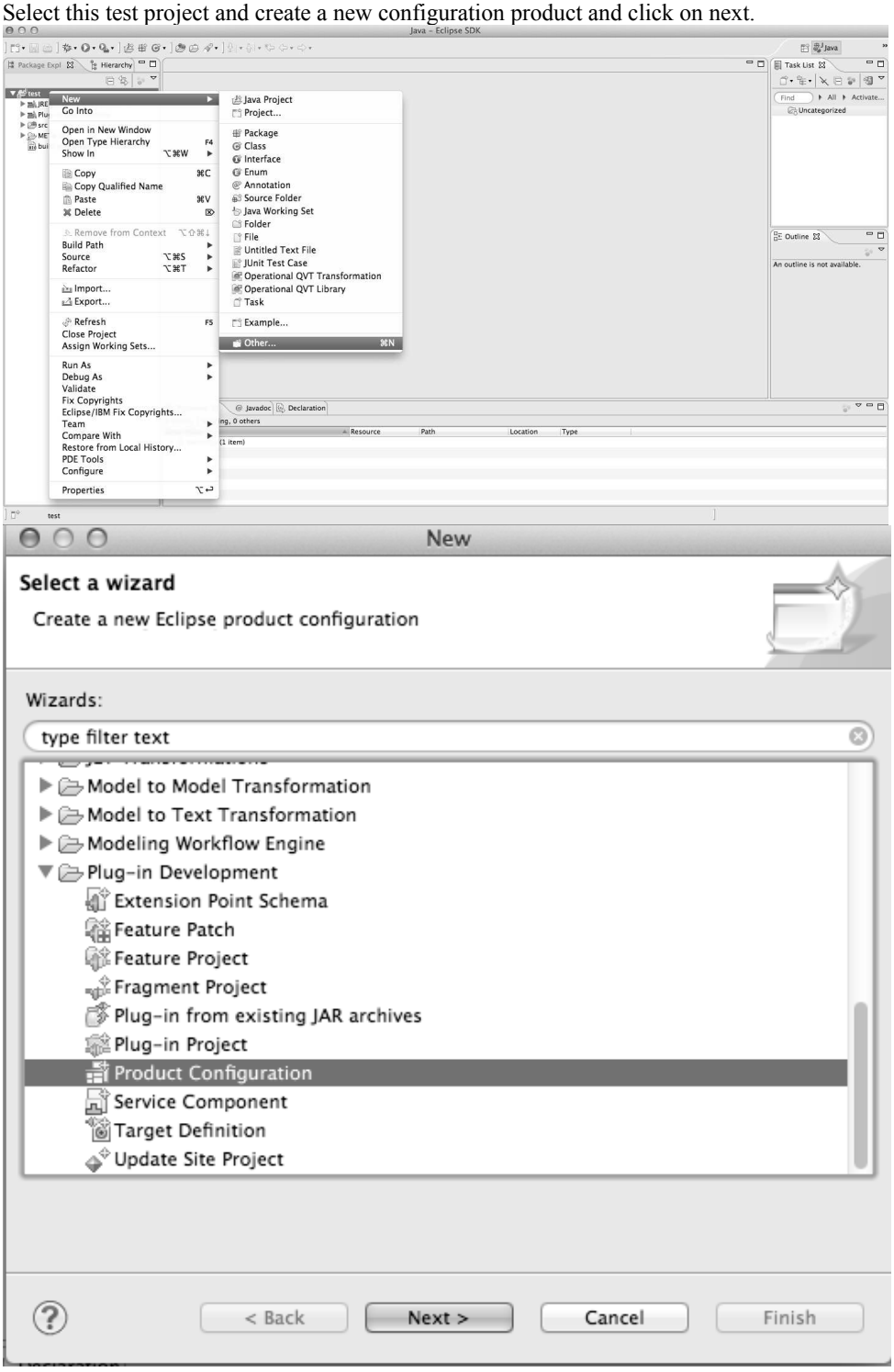


4. A new instance of Eclipse is launched. In this new instance create a new Java project (here called “test”)

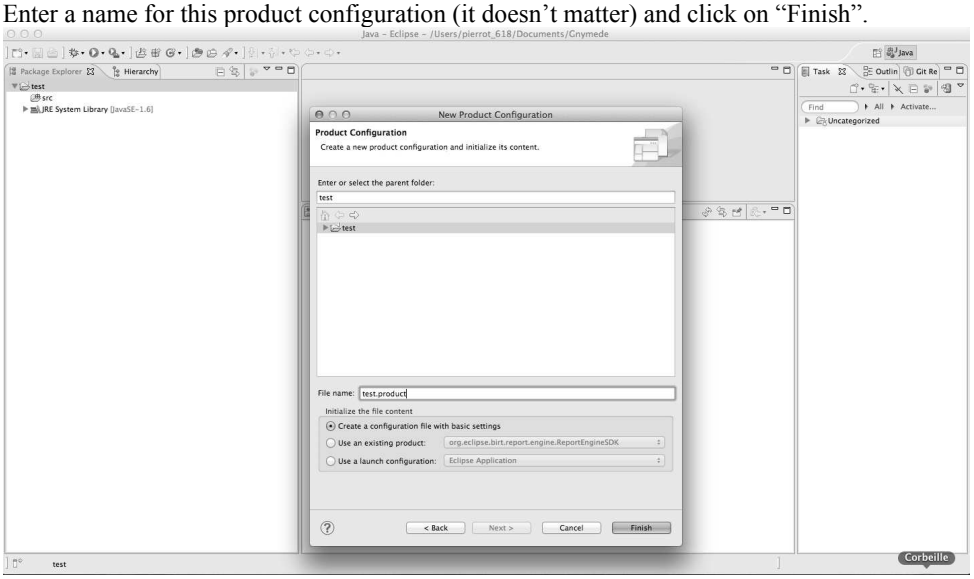




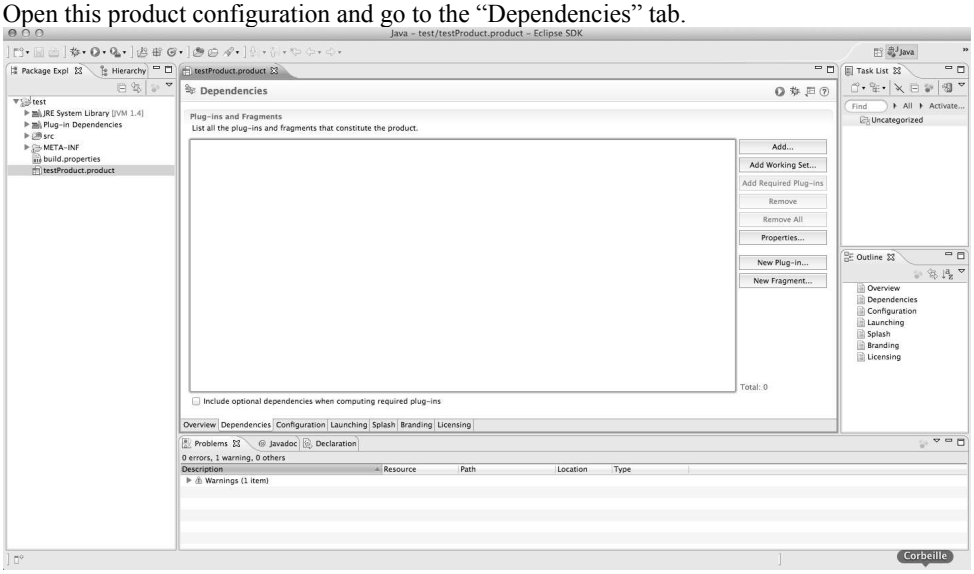
5. Select this test project and create a new configuration product and click on next.



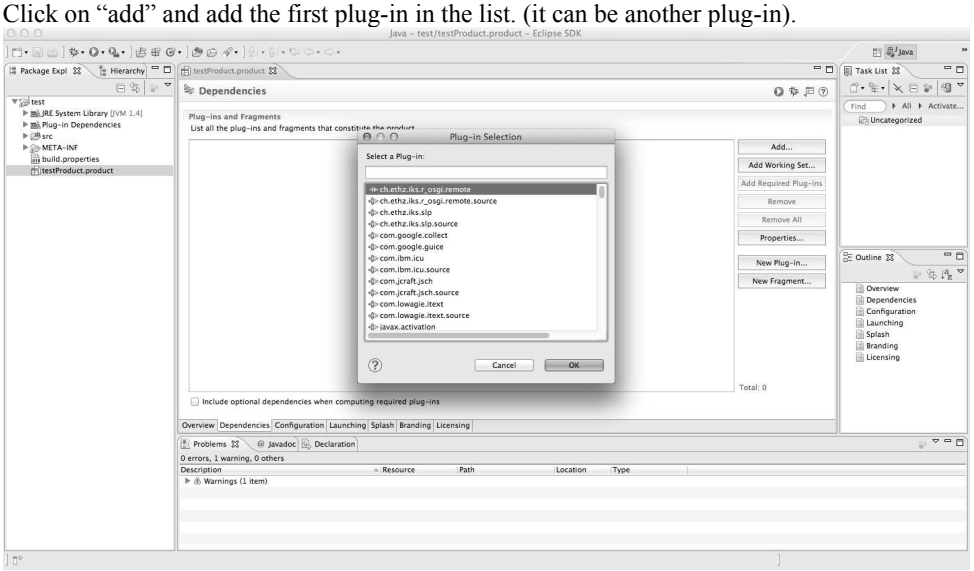
6. Enter a name for this product configuration (it doesn't matter) and click on "Finish".



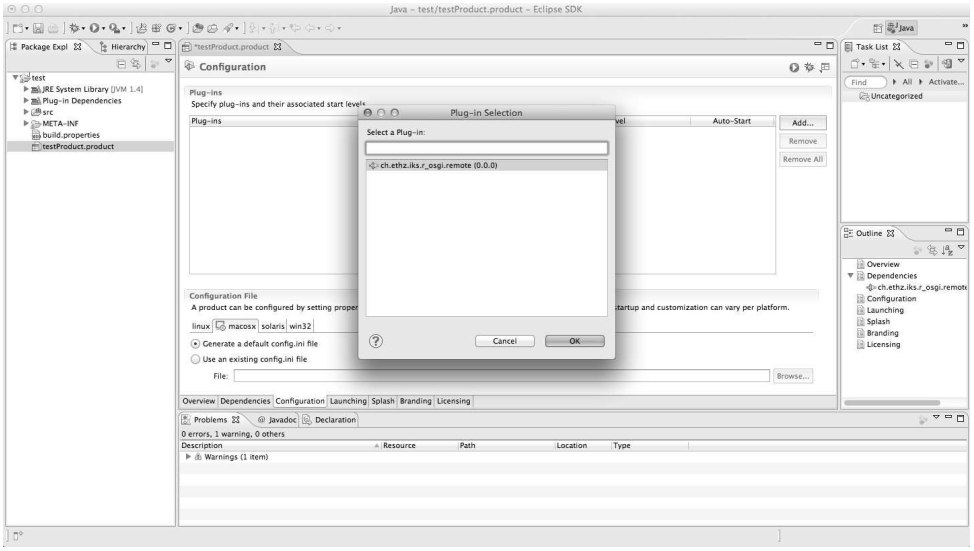
7. Open this product configuration and go to the "Dependencies" tab.



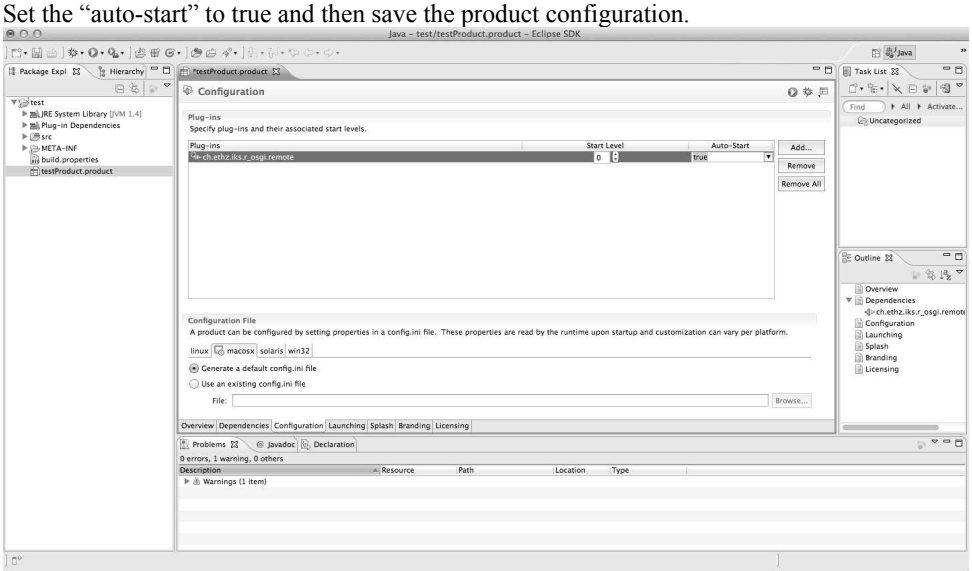
8. Click on “add” and add the first plug-in in the list. (it can be another plug-in).



9. Click “OK”.
10. Go to “Configuration” tab and click on “add “. Select the plug-in in the list and click on “OK”.



11. Set the “auto-start” to true and then save the product configuration.



12. Go by the file explorer of the computer (not in eclipse) at the same level of the workspace of Eclipse (Desktop -> runtimeEclipseApplication) and search the folder “runtime-EclipseApplication”. Open it and you will find your “test” project. Open it and open the file “.product” with a text editor. Look at the value of the variable “auto-start” and here is the
Bug !

```
<?xml version="1.0" encoding="UTF-8"?>
<?pde version="3.5"?>

<product useFeatures="false">

   <configIni use="default">
   </configIni>

   <launcherArgs>
      <vmArgsMac>-XstartOnFirstThread -Dorg.eclipse.swt.internal.carbon.smallFonts</vmArgsMac>
   </launcherArgs>

   <plugins>
      <plugin id="ch.ethz.iks.r_osgi.remote"/>
   </plugins>

   <configurations>
      <plugin id="ch.ethz.iks.r_osgi.remote" autoStart="false" startLevel="0" />
   </configurations>

</product>
```

13. The right version has to be.

```
<?xml version="1.0" encoding="UTF-8"?>
<?pde version="3.5"?>

<product useFeatures="false">

   <configIni use="default">
   </configIni>

   <launcherArgs>
      <vmArgsMac>-XstartOnFirstThread -Dorg.eclipse.swt.internal.carbon.smallFonts</vmArgsMac>
   </launcherArgs>

   <plugins>
      <plugin id="ch.ethz.iks.r_osgi.remote"/>
   </plugins>

   <configurations>
      <plugin id="ch.ethz.iks.r_osgi.remote" autoStart="true" startLevel="0" />
   </configurations>

</product>
```

14. Close the launched Eclipse and go back to source code.

C.3 jEdit

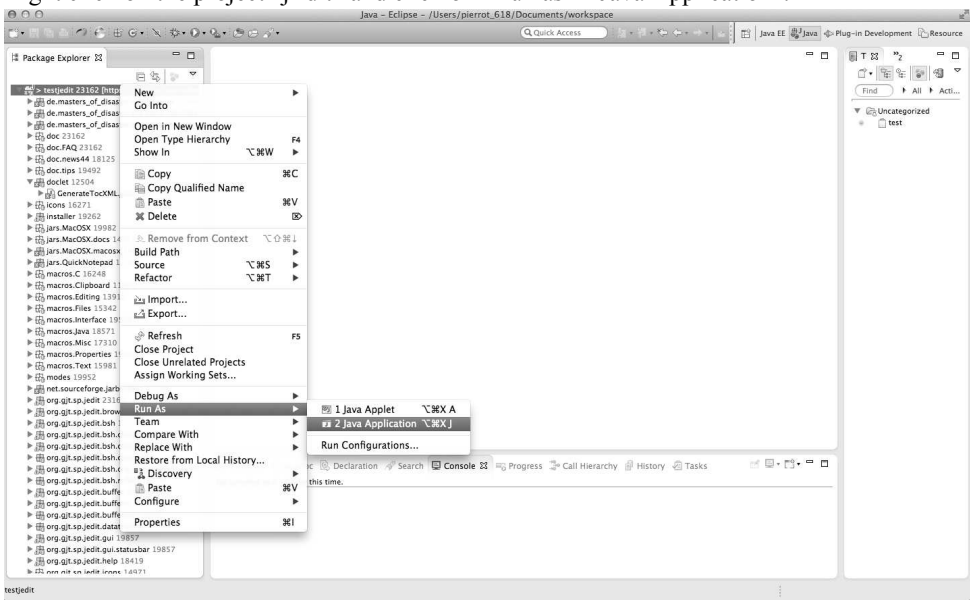
Task 5: Presentation

Task name

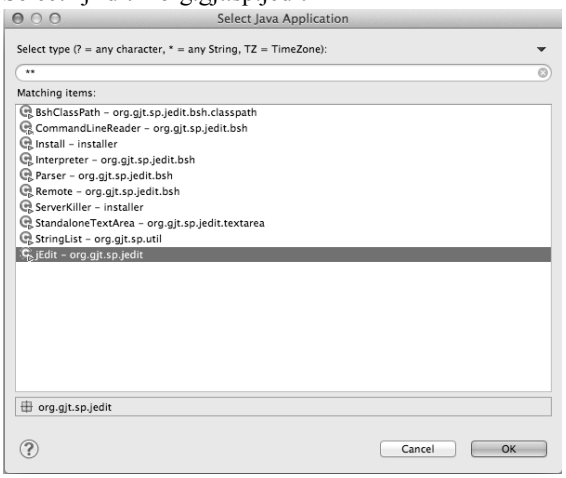
Add lines number and error messages in the select line range dialog

Steps-to-reproduce

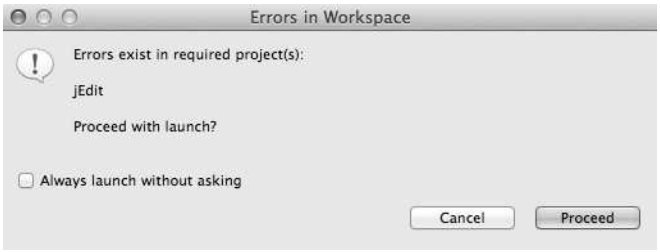
1. Call the experimenter
2. Right click on the project “jEdit” and click on “Run as -> Java Application”.



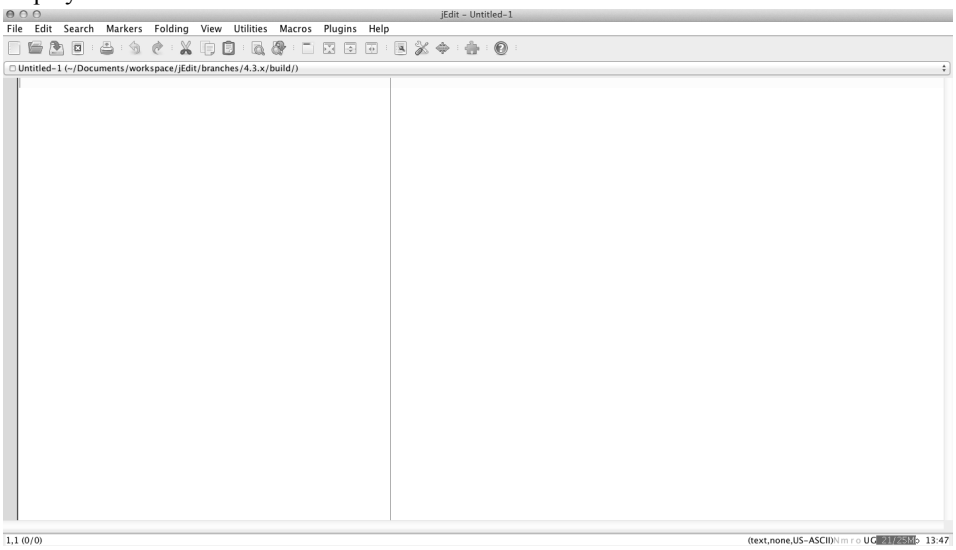
3. Select “jEdit – org.gjt.sp.jedit”



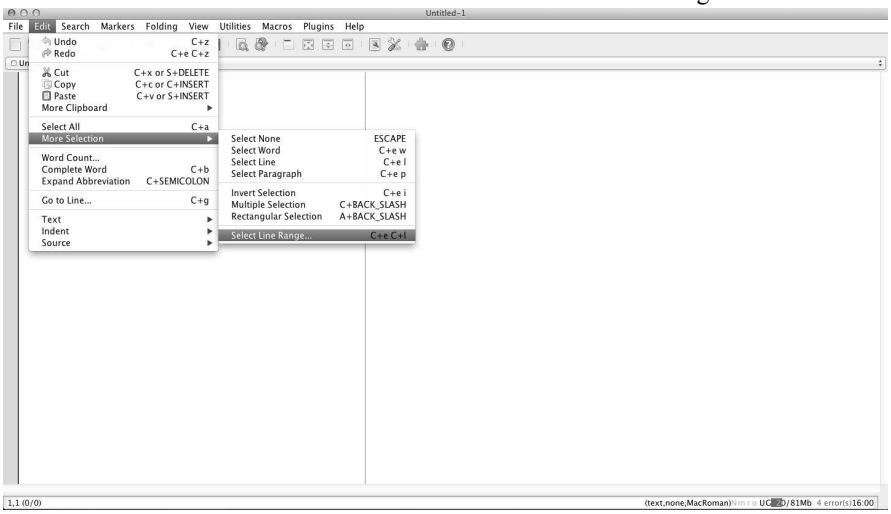
4. Ignore errors and click on “proceed”. The jEdit Application will be launched.



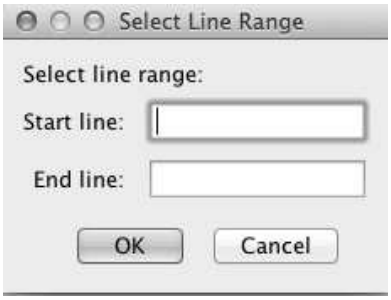
5. Once jEdit is launched you will have this page displayed



6. On the menuBar lick on “Edit -> More selection -> Select Line Range”



7. This popup will open and here is the bug. You have to display next to the “Select line range:” label the number of lines contained in the file (example: (1-15)) and if the user enter a wrong character (everything that’s not a number) or a wrong interval you have to display a message if the user press the “Ok” button.



8. Close the “jEdit instance”

C.4 JHotDraw

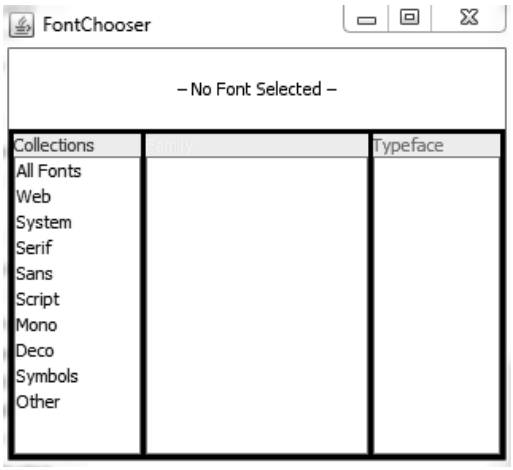
Task 6: Presentation

Task name

Change colors of labels and background of the “FontChooser”.

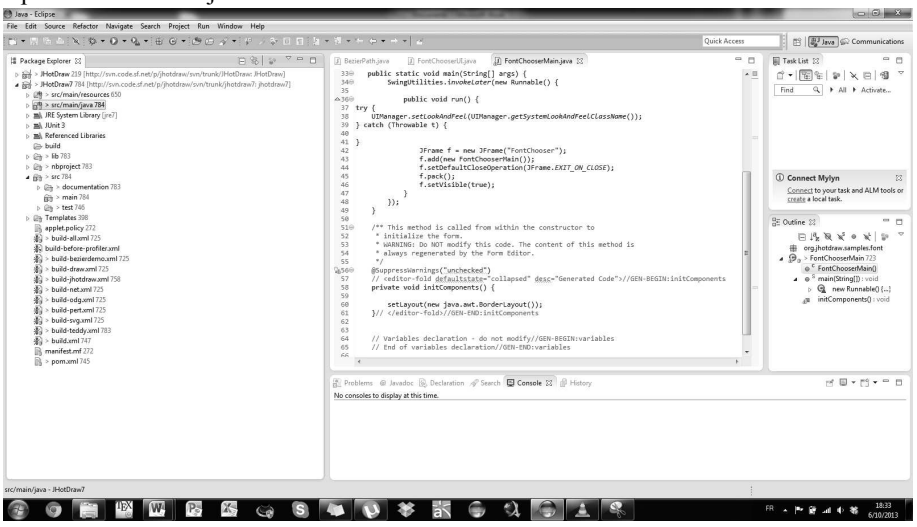
Description

- The "Collections" label will be in blue, the "Family" label in yellow and the "Typeface" label in red.
- The background of the window will be in black.

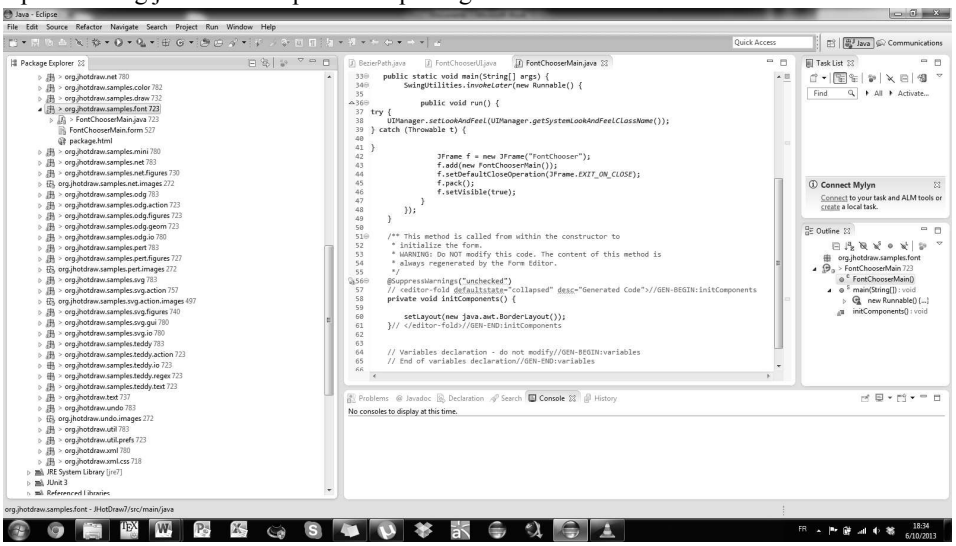


Steps-to-reproduce

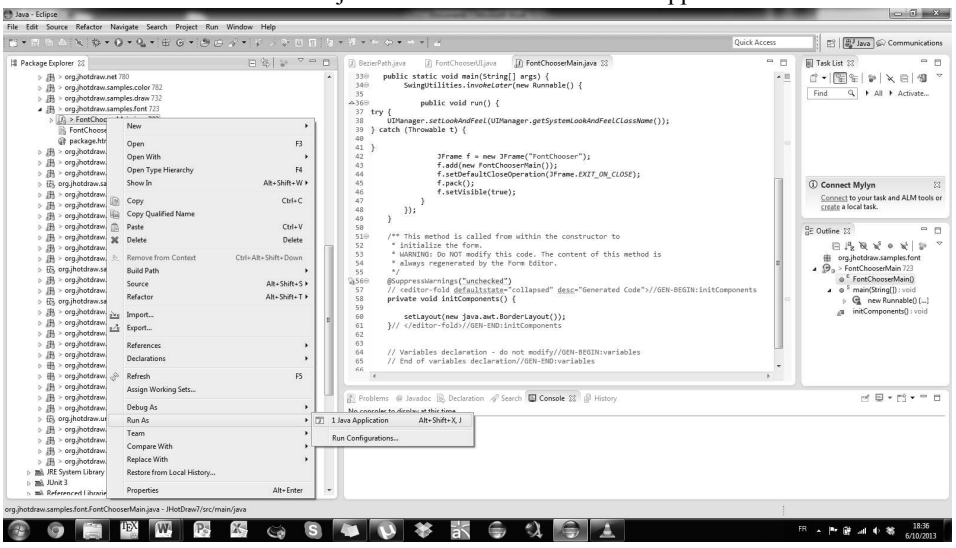
1. Call the experimenter.
2. Open the “src/main/java folder”.



3. Open the “org.jhotdraw.samples.font” package.

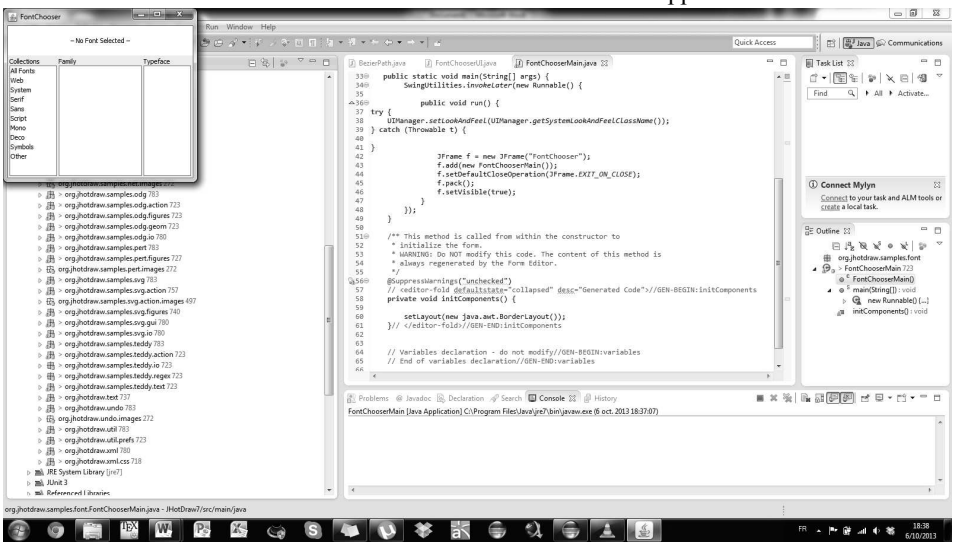


4. Click on the “FontChooserMain.java” class and run it as a Java application.



5. At this point, an error box will appears, just press “Proceed”.

6. Wait some time and a little window named “FontChooser” will appear.



7. Exit the FontChooserApplication.

D Checklist

Sujet : _____

Checklist

Quand le sujet arrive :

Accueillir le sujet	
Donner un numéro identifiant	
Donner un poste de travail	
Donner une petite explication de l'expérience	
Faire remplir le pré-questionnaire	
Définir la tâche du sujet sur bas du pré-questionnaire	
Expliquer la procédure de l'expérience	
Lancer la capture vidéo.	
Démarrer sa version d'Eclipse	
Insister sur le fait qu'il peut stopper à tout moment et qu'il peut nous poser des questions pendant l'expérience tant que ce n'aide pas à résoudre le bug.	

Quand le sujet a fini :

Demander s'il est sûr d'avoir fini	
Enregistrer la capture vidéo	
Exporter son contexte Mylyn dans le dossier Raw	
Renommer le contexte par « NomDuProjet_NumDeLaTâche_NuméroIdentifiant.zip »	
Extraire le contexte local (context -> local1.xml)et le placer dans le dossier inIH	
Renommer le contexte local par « NomDuProjet_NumDeLaTâche_NuméroIdentifiant.xml »	
Créer le patch (team -> create patch) avec le nom « NomDuProjet_NumDeLaTâche_NuméroIdentifiant.patch »	
Placer le patch dans le dossier inPatch	
Lancer le compute.sh dans un terminal	

Quand le sujet a fini :

Remercier le sujet	
Supprimer la tâche	
Remplacer le workspace avec un « replace with the lastest form repository »	
Créer une nouvelle tâche	
Supprimer les fichiers « Eclipse runtime application »	
Accueillir le prochain sujet	

E Procédure de l'expérimentation

Experiment Description and Procedure

Goal

This experiment aims to evaluate the relevancy and usefulness of program entities e.g., how developers identify the program entities that need to be changed to perform a task, and what are the program entities that are needed to understand the program.

Experiment Description

Using Eclipse IDE you will perform one maintenance task on a Java project. We will provide a quick description of the project. We have designed this experiment to be as short and simple as possible. The estimate time to perform the task is about 45min but there is no time limit. Please try your best to complete the task. The information we gain from this experiment is totally anonymous. You can leave the experiment at any time for any reason without penalty of any kind.

Experiment Procedure

For this experiment, you will:

1. Fill the pre-experiment questionnaire.
2. Read the description of the project you have to work with.
3. Read the description of your task then follow the instructions given to you by the experimenter.
4. Perform the task
    a) The experimenter will show you how to activate your task. Activate the task and start performing it.
    b) When you think you have finished your task, deactivate the task. **If you deactivate the task without having completed it, you can still activate it and continue to perform the task later. We strongly recommend you to deactivate the task when you think that you have completely addressed it.**
5. Let the experimenter know that you have finished your task.
6. Going through the steps that you followed to perform the task, we will be asking you some questions.
7. Give us your feedback/comments on the experiment.

F Configuration d'Eclipse

F.1 Eclipse 3.5.2

NAME	VERSION	ID
Dynamic Languages Toolkit - Mylyn Integration	1.0.0.v20090610-1638-1--7w311_17261119	org.eclipse.dltk.mylyn.feature.group
Eclipse EGit	2.1.0.201209190230-r	org.eclipse.egit.feature.group
Eclipse EGit - Source	2.1.0.201209190230-r	org.eclipse.egit.source.feature.group
Eclipse JGit	2.1.0.201209190230-r	org.eclipse.jgit.feature.group
Eclipse JGit - Source	2.1.0.201209190230-r	org.eclipse.jgit.source.feature.group
Eclipse Platform	3.5.2.M20100211-1343	org.eclipse.platform.ide
Eclipse Platform SDK	3.5.2.M20100211-1343	org.eclipse.platform.sdk
Eclipse Releng Tools	3.3.0.v20090520-4407S46yaw311A1923	org.eclipse.releng.tools.feature.group
Eclipse SDK	3.5.2.M20100211-1343	org.eclipse.sdk.ide
Environment Description for CDC-1.0/Foundation-1.0	1.0.0.v20090407-1030	org.eclipse.pde.api.tools.ee.cdcfoundation10
Environment Description for CDC-1.1/Foundation-1.1	1.0.0.v20090407-1030	org.eclipse.pde.api.tools.ee.cdcfoundation11
Environment Description for J2SE-1.2	1.0.0.v20090407-1030	org.eclipse.pde.api.tools.ee.j2se12
Environment Description for J2SE-1.3	1.0.0.v20090407-1030	org.eclipse.pde.api.tools.ee.j2se13
Environment Description for J2SE-1.4	1.0.0.v20090407-1030	org.eclipse.pde.api.tools.ee.j2se14
Environment Description for J2SE-1.5	1.0.0.v20090407-1030	org.eclipse.pde.api.tools.ee.j2se15
Environment Description for JavaSE-1.6	1.0.0.v20090407-1030	org.eclipse.pde.api.tools.ee.javase16
Environment Description for JRE-1.1	1.0.0.v20090407-1030	org.eclipse.pde.api.tools.ee.jre11
Environment Description for OSGi/Minimum-1.0	1.0.0.v20090407-1030	org.eclipse.pde.api.tools.ee.osgiminimum10
Environment Description for OSGi/Minimum-1.1	1.0.0.v20090407-1030	org.eclipse.pde.api.tools.ee.osgiminimum11
Environment Description for OSGi/Minimum-1.2	1.0.1.v20090407-1030	org.eclipse.pde.api.tools.ee.osgiminimum12
Mylyn Bridge: C/C++ Development	5.1.0.201002161416	org.eclipse.cdt.mylyn.feature.group
Mylyn Bridge: Eclipse IDE	3.2.3.v20100217-0100-e3x	org.eclipse.mylyn.ide_feature.feature.group
Mylyn Bridge: Java Development	3.2.3.v20100217-0100-e3x	org.eclipse.mylyn.java_feature.feature.group
Mylyn Bridge: Plug-in Development	3.2.3.v20100217-0100-e3x	org.eclipse.mylyn.pde_feature.feature.group
Mylyn Bridge: Team Support	3.2.3.v20100217-0100-e3x	org.eclipse.mylyn.team_feature.feature.group
Mylyn Connector: Bugzilla	3.2.3.v20100217-0100-e3x	org.eclipse.mylyn.bugzilla_feature.feature.group
Mylyn Task List (Required)	3.2.3.v20100217-0100-e3x	org.eclipse.mylyn_feature.feature.group
Mylyn Task-Focused Interface (Recommended)	3.2.3.v20100217-0100-e3x	org.eclipse.mylyn.context_feature.feature.group
Mylyn WikiText	1.1.3.v20100217-0100-e3x	org.eclipse.mylyn.wikitext_feature.feature.group
PDE/API Tools Environment Descriptions	1.0.0.v20090512-7C-0F9JgLMsM0MXAWLrL	org.eclipse.pde.api.tools.ee.fragments.feature.jar
Subversive SVN Connectors	2.2.1.I20091009-1900	org.polarion.eclipse.team.svn.connector.feature.group
Subversive SVN Integration for the Mylyn Project (Optional) (Incubation)	0.7.8.I20091023-1300	org.eclipse.team.svn.mylyn.feature.group
SVNKit 1.3.0 Implementation (Optional)	2.2.1.I20091009-1900	org.polarion.eclipse.team.svn.connector.svnkit16.feature.group

F.2 Eclipse 4.3

NAME	Version	ID
ECF Target Components for Eclipse	3.7.1.v20131027-1505	org.eclipse.ecf.core.feature.group
Eclipse Plug-in Development Environment	3.9.1.v20130911-1000	org.eclipse.pde.feature.group
Eclipse Standard/SDK	2.0.0.20130613-0530	epp.package.standard
Mylyn Builds Connector: Hudson/Jenkins	1.1.1.v20130917-0100	org.eclipse.mylyn.hudson.feature.group
Mylyn Context Connector: C/C++ Development	5.5.1.v20130917-0100	org.eclipse.cdt.mylyn.feature.group
Mylyn Context Connector: Ecore Tools (Incubation)	0.9.2.I20130903-2034	org.eclipse.mylyn.mft.ecoretools.feature.group
Mylyn Context Connector: EMF (Incubation)	0.9.2.I20130903-2034	org.eclipse.mylyn.mft.emf.feature.group
Mylyn Context Connector: GMF (Incubation)	0.9.2.I20130903-2034	org.eclipse.mylyn.mft.gmf.feature.group
Mylyn Context Connector: Java Development	3.9.1.v20130917-0100	org.eclipse.mylyn.java_feature.feature.group
Mylyn Context Connector: Papyrus UML (Incubation)	0.9.2.I20130903-2034	org.eclipse.mylyn.mft.papyrus.feature.group
Mylyn Context Connector: Plug-in Development	3.9.1.v20130917-0100	org.eclipse.mylyn.pde_feature.feature.group
Mylyn Intent (Incubation)	0.8.1.201308291003	org.eclipse.mylyn.docs.intent.feature.feature.group
Mylyn Intent - CDO support (Incubation)	0.8.0.201306030818	org.eclipse.mylyn.docs.intent.cdo.feature.feature.group
Mylyn Intent - Workspace Support (Incubation)	0.8.1.201308291003	org.eclipse.mylyn.docs.intent.workspace.feature.feature.group
Mylyn Intent Connector: Java (Incubation)	0.8.1.201308291003	org.eclipse.mylyn.docs.intent.bridge.java.feature.feature.group
Mylyn Intent Generator: Intent Documentation (Incubation)	0.8.1.201308291003	org.eclipse.mylyn.docs.intent.exporter.feature.feature.group
Mylyn Reviews Connector: Gerrit	2.0.1.v20130917-0100	org.eclipse.mylyn.gerrit.feature.feature.group
Mylyn Tasks Connector: Trac	3.9.1.v20130917-0100	org.eclipse.mylyn.trac_feature.feature.group
Mylyn Versions Connector: CVS	1.1.1.v20130917-0100	org.eclipse.mylyn.cvs.feature.group
Mylyn Versions Connector: Git	1.1.1.v20130917-0100	org.eclipse.mylyn.git.feature.group
Mylyn WikiText	1.8.1.v20130917-0100	org.eclipse.mylyn.wikitext_feature.feature.group
Mylyn WikiText: Additional Generators (Incubation)	0.8.1.201308291003	org.eclipse.mylyn.docs.intent.markup.feature.feature.group
Source for ECF Target Components for Eclipse	3.7.1.v20131027-1505	org.eclipse.ecf.core.source.feature.group
Subversive SVN Connectors	3.0.2.I20130808-1700	org.polarion.eclipse.team.svn.connector.feature.group
Subversive SVN Integration for the Mylyn Project (Optional)	1.1.0.I20130619-1700	org.eclipse.team.svn.mylyn.feature.group
Subversive SVN Team Provider	1.1.1.I20130816-1700	org.eclipse.team.svn.feature.group
Subversive SVN Team Provider Sources	1.1.1.I20130816-1700	org.eclipse.team.svn.source.feature.group
SVNKit 1.7.10 Implementation (Optional)	3.0.2.I20130808-1700	org.polarion.eclipse.team.svn.connector.svnkit17.feature.group

G Données sur les sujets